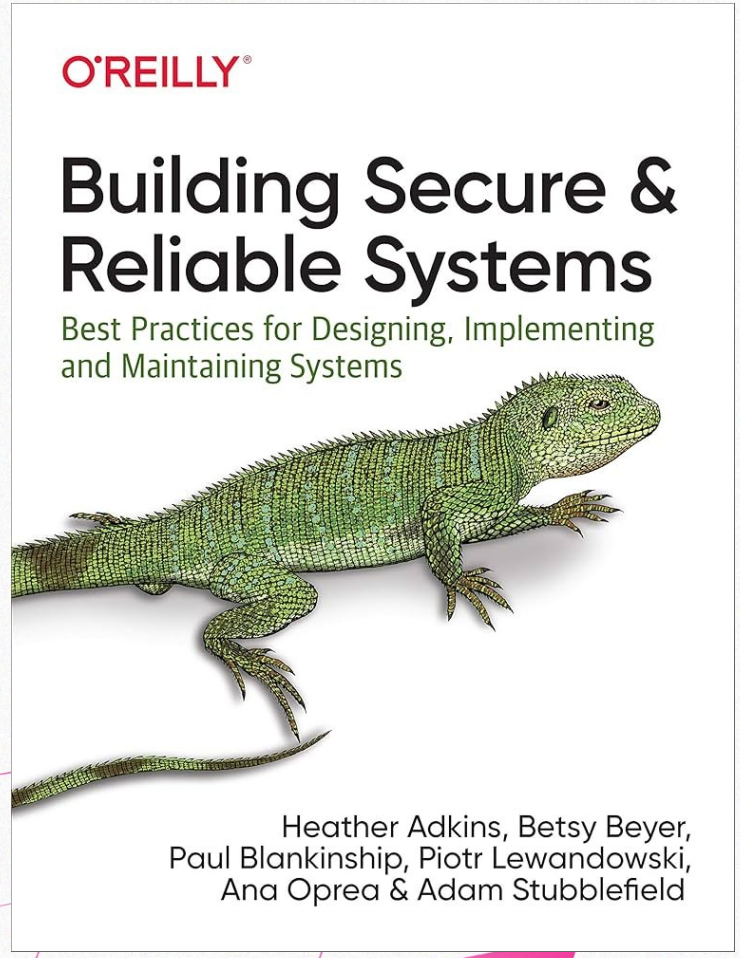
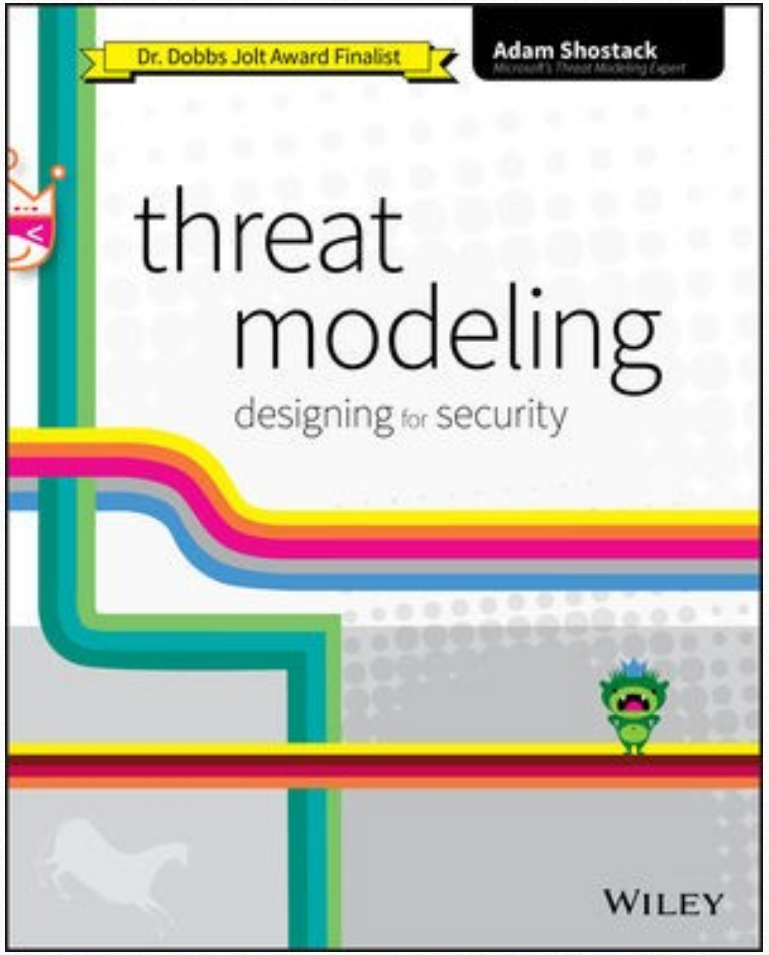
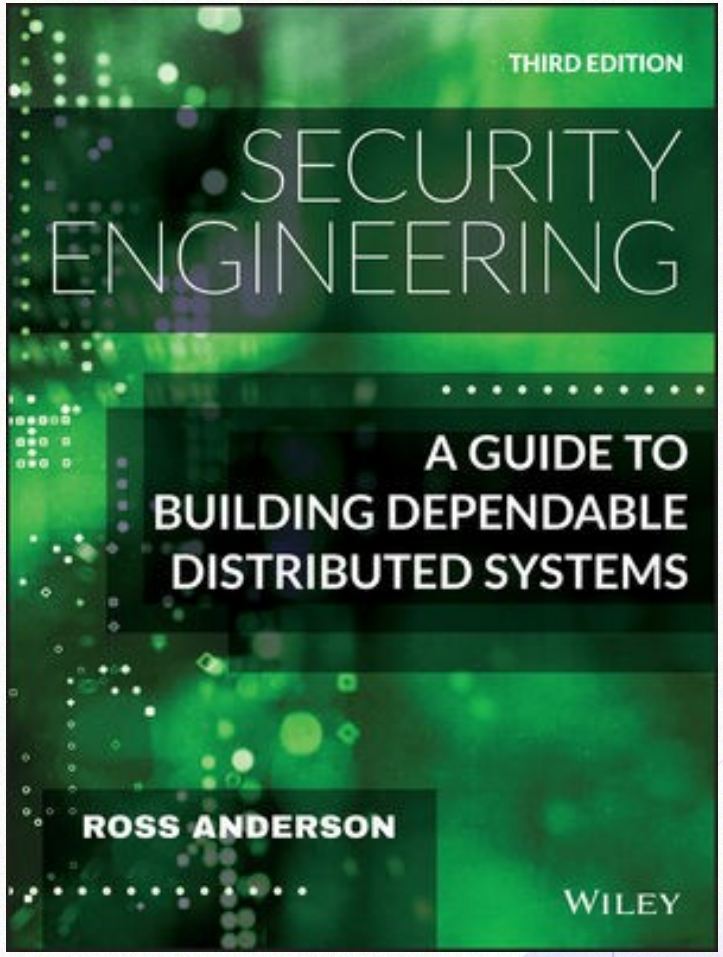
Introducción a la Seguridad en la Arquitectura de Software

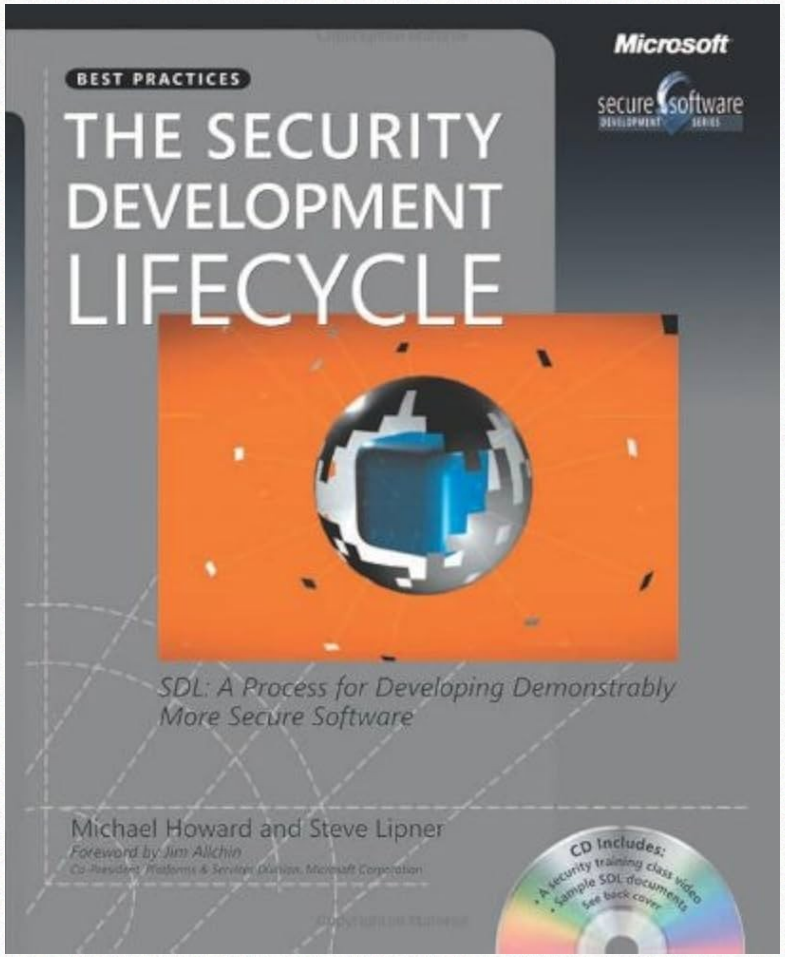
“El diseño arquitectónico de software es la clave para construir sistemas sostenibles.”

— Grady Booch


O'REILLY
Building Secure & Reliable Systems
Best Practices for Designing, Implementing and Maintaining Systems
Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea & Adam Stubblefield


Dr. Dobbs Jolt Award Finalist
Adam Shostack
threat modeling
designing for security
WILEY


THIRD EDITION
SECURITY ENGINEERING
A GUIDE TO BUILDING DEPENDABLE DISTRIBUTED SYSTEMS
ROSS ANDERSON
WILEY


BEST PRACTICES
Microsoft
secure software
THE SECURITY DEVELOPMENT LIFECYCLE
SDL: A Process for Developing Demonstrably More Secure Software
Michael Howard and Steve Lipner
Foreword by Jim Allchin
CD Includes:
• A security training class video
• Sample SDL documents
See back cover

Imaginemos un sistema bancario basado en microservicios, con millones de transacciones diarias.

¿Dónde pondrías el primer ‘muro’ de seguridad?

Fundamentos de seguridad en arquitectura

Confidencialidad: proteger la información de accesos no autorizados.

Integridad: asegurar que los datos no se modifiquen sin control.

Disponibilidad: garantizar el acceso de usuarios legítimos cuando lo necesiten.

Si tuvieras que sacrificar una de las tres propiedades (Confidencialidad, Integridad, Disponibilidad) para mejorar las otras dos, ¿cuál sería y en qué contexto?

Confidencialidad

Si un mensaje viaja cifrado, ¿quién debe desencriptarlo y cuándo?

Capas de cifrado

“¿Qué capas de cifrado tienes implementadas: TLS en transporte, cifrado a nivel de campo, cifrado de base de datos?”

“Si sólo cifras en reposo, ¿cómo previenes la exposición durante la transmisión interna?”

Gestión de llaves

“¿Dónde y cómo almacenan las claves maestras? ¿Usan un KMS gestionado, un HSM local o un vault interno?”

“Si pierden la llave maestra, ¿qué plan de recuperación o rotación de claves tienen?”

Tokenización y anonimización

“¿Qué datos tokenizan o anonimizan antes de guardar logs o enviarlos a sistemas analíticos?”

“¿Prefieren tokenización reversible o irreversible? ¿En qué casos usarían cada una?”

Acceso y auditoría

“¿Quién tiene permisos para descifrar datos y en qué situaciones? ¿Cómo registran cada acceso?”

“¿Tienen roles separados para desarrolladores y equipos forenses? ¿Cómo verifican que nadie abuse de sus permisos?”

Integridad

Supongamos que una solicitud de pago por 1,000 USD viaja desde el frontend hasta el motor de pagos.

Si alguien altera el campo amount, ¿cómo detectamos esa modificación?

Firma y validación
"¿Usan checksums o HMAC? ¿Por qué?"

"¿En qué punto validan la firma: en un API Gateway, en cada microservicio o en un proxy central?"

Protección contra replay

"Si emplean Kafka para mensajes, ¿cómo evitan ataques de replay en los topics de transacciones?"

"¿Agregan timestamps o nonces, o emplean otro mecanismo?"

Modelos avanzados

"¿Han considerado un ledger distribuido para garantizar inmutabilidad?"

"¿Qué impacto tendría en la latencia y el costo operativo?

Disponibilidad

La organización exige 99.999% de uptime.

Si ciframos de extremo a extremo, agregamos latencia. ¿Cómo coordinan esa tensión con el equipo de operaciones?

Estrategias de replicación

"¿Prefieren réplicas en varias regiones o escalado horizontal en una sola zona?"

"¿Cómo gestionan la consistencia entre réplicas?"

Plan de degradación

"¿Qué funcionalidades podrían degradarse primero ante un pico de carga para mantener la seguridad de las más críticas?"

"¿Cómo comunican esa degradación a los usuarios?"

Impacto de la seguridad en el SLA

"¿Cómo miden el efecto del cifrado intenso en los tiempos de respuesta?"

"¿Tienen métricas específicas o alertas en el pipeline de CI/CD?"

¿Qué es la Seguridad en el Diseño Arquitectónico?

Cuando hablamos de seguridad en el diseño arquitectónico, nos referimos a los principios y prácticas que minimizan las vulnerabilidades en un sistema desde su concepción. Esto incluye cómo los componentes interactúan, qué datos procesan y cómo manejan errores.

La arquitectura no es solo un plano técnico; es un **sistema de defensa** que protege contra amenazas externas e internas.

Principio de Mínima Exposición

Imaginemos que cada componente de un sistema tiene puertas. El principio de mínima exposición dice que estas puertas deben mantenerse cerradas, y solo abrirse cuando sea absolutamente necesario.

Ejemplo práctico:

En una arquitectura de microservicios, solo exponemos las APIs necesarias y protegemos otras funcionalidades internas.

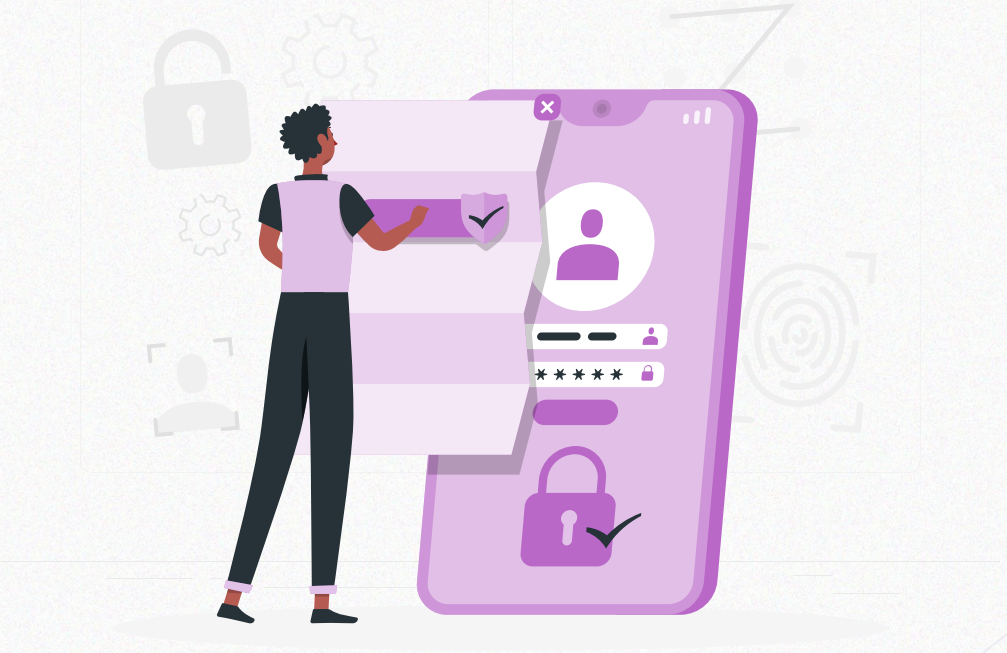
Principio de Control de Acceso Granular

Este principio asegura que los usuarios o sistemas solo tengan acceso a lo que realmente necesitan, ni más ni menos.

Ejemplo práctico:

- Usuario A: Solo puede leer datos.
- Usuario B: Puede leer y escribir datos.
- Usuario C: No tiene acceso a los datos.

Los sistemas deben implementar **roles y permisos específicos**. Por ejemplo, un empleado de recursos humanos no necesita acceso a los datos financieros de la empresa.



Principio de Protección de Datos Sensibles

No toda la información tiene el mismo valor, pero algunos datos, como contraseñas, información personal o datos bancarios, son extremadamente sensibles y necesitan protección adicional.

Prácticas clave:

• **Cifrado**: Si un atacante obtiene una base de datos, los datos deben ser ilegibles (p. ej., cifrar contraseñas con bcrypt).

• **Máscara de datos**: Mostrar solo parte de los datos, como “**-**-1234” en números de tarjetas de crédito.

Principio de Diseño Asumido como Inseguro

Un error común es asumir que un sistema es seguro. Este principio nos obliga a diseñar como si siempre existiera una amenaza activa.

Ejemplo práctico:

Una API pública debe validarse para evitar inyección de código o accesos indebidos, aunque solo esperemos clientes legítimos.

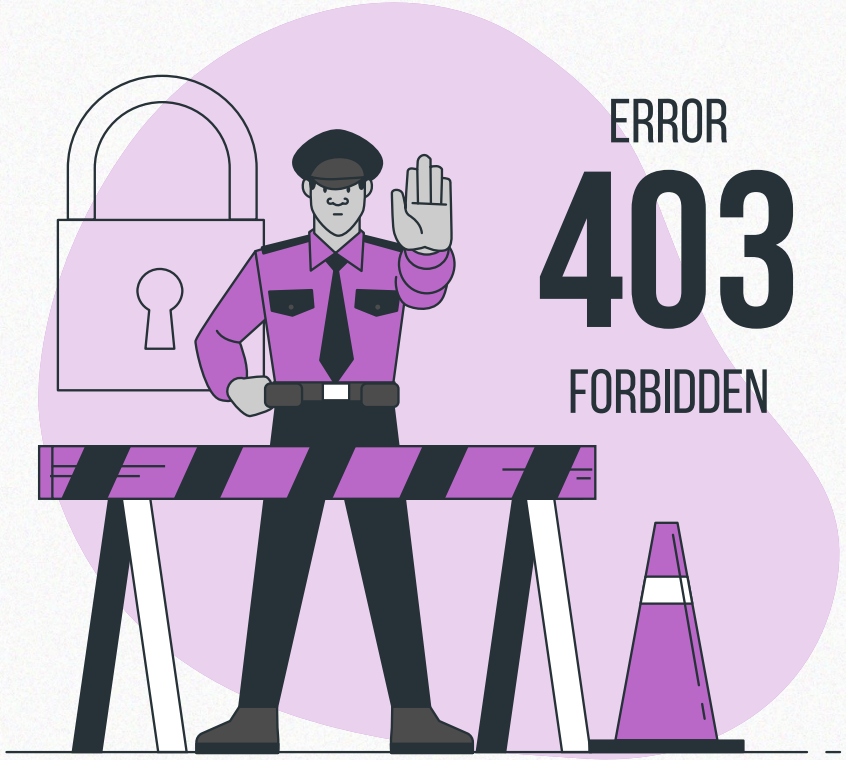
Principio de Falla Segura

Los sistemas deben estar diseñados para fallar de manera que no expongan información sensible ni comprometan la seguridad.

Ejemplo práctico:

• Si un usuario excede intentos fallidos al iniciar sesión, bloquea la cuenta temporalmente en lugar de permitir intentos ilimitados.

• Cuando un sistema pierde conexión con un servidor crítico, asegura que los datos sensibles no sean accesibles.

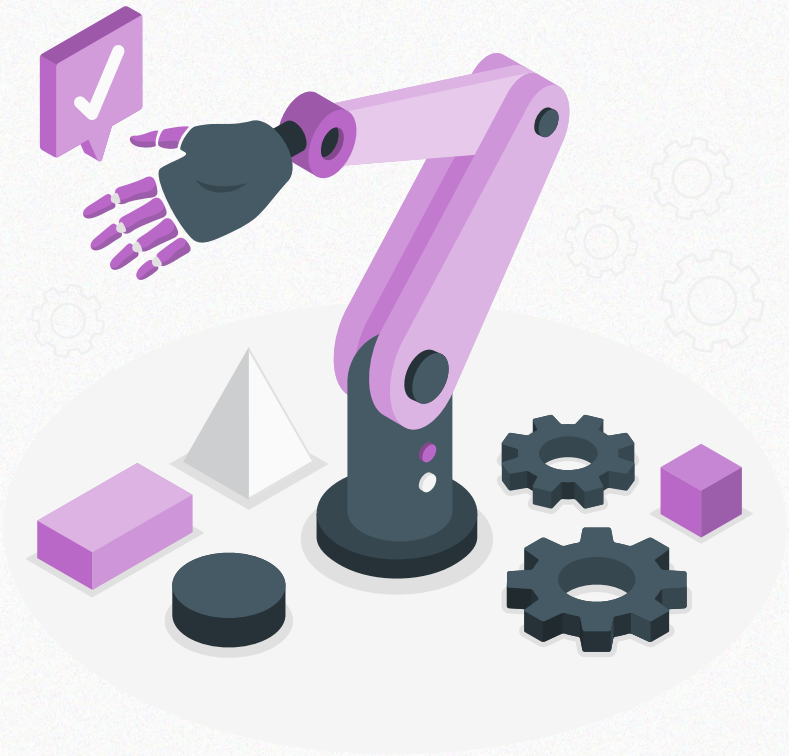


Economía de Mecanismos

Los sistemas más simples tienden a ser más seguros porque es más fácil analizar, probar y mantener sus mecanismos.

Ejemplo práctico:

Un sistema de autenticación simple y bien probado (como OAuth2) es mejor que uno casero y complejo.



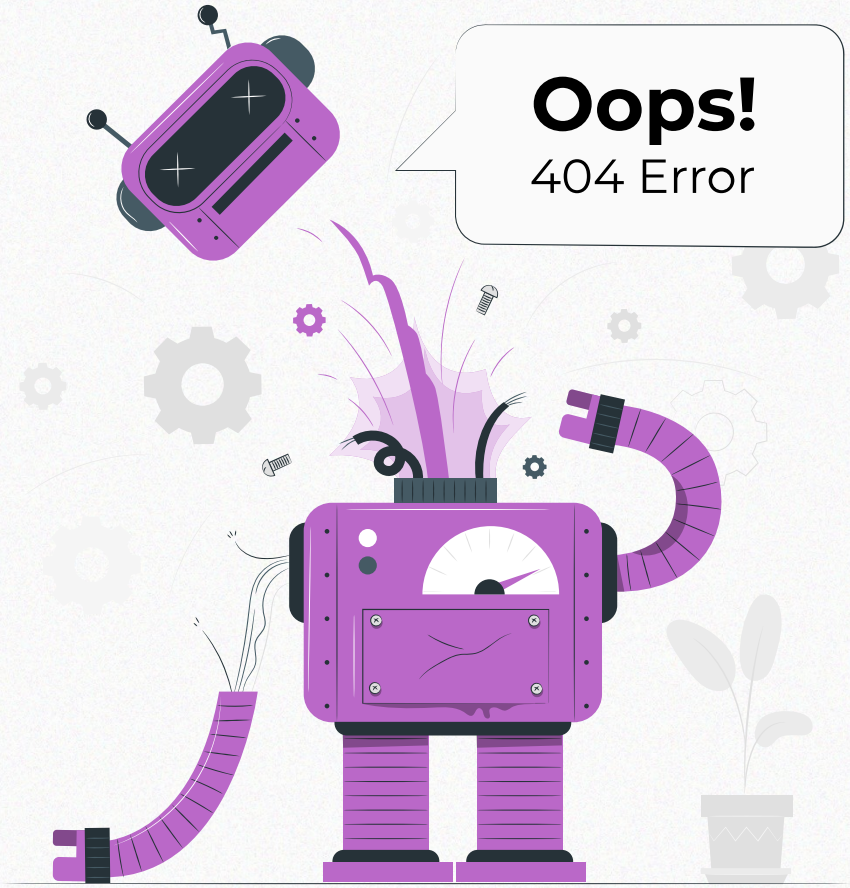
Ejercicio Practico

Diseñar un sistema de inicio de sesión para una aplicación bancaria.

Caso de estudio

Investigar un caso de estudio de una brecha de seguridad reciente y analizar qué principio de diseño no se respetó.

¿Cómo podría haberse prevenido?



OWA SP

Open Web Application Security Project

Es una fundación sin fines de lucro dedicada a mejorar la seguridad del software. Desde su creación en 2001, se ha convertido en una referencia global para el desarrollo seguro de aplicaciones, ofreciendo herramientas, documentación, estándares y recursos de código abierto.

Objetivos Principales de OWASP:

1. **Concienciar sobre la seguridad del software:** Difundir la importancia de desarrollar aplicaciones seguras.

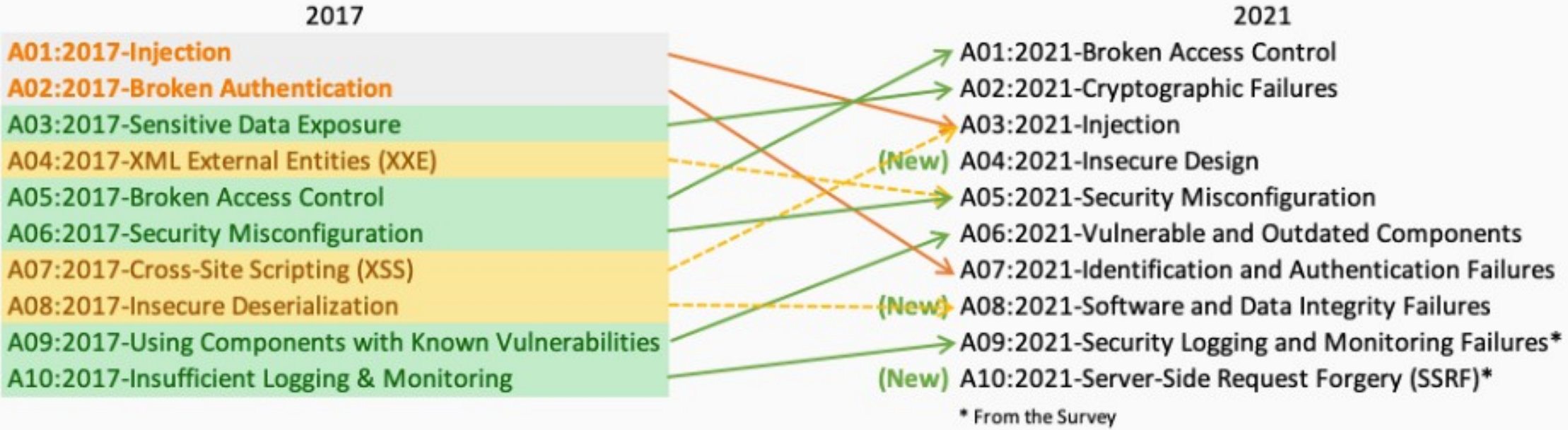
2. **Proveer recursos gratuitos:** Desarrollar herramientas, documentación y guías prácticas para mejorar la seguridad en el desarrollo de software.

3. **Fomentar una comunidad activa:** OWASP cuenta con una red global de voluntarios, capítulos locales y eventos como conferencias y talleres.

OWASP
Open Web Application Security Project

OWASP Top Ten

Lista de las 10 vulnerabilidades más críticas en aplicaciones web, actualizada regularmente.
Es ampliamente utilizada como referencia en auditorías y desarrollos seguros.



OWASP Top Ten

Lista de las 10 vulnerabilidades más críticas en aplicaciones web, actualizada regularmente.

Es ampliamente utilizada como referencia en auditorías y desarrollos seguros.

1. Inyección (SQL, OS, LDAP...)
2. Autenticación rota
3. Exposición de datos sensibles
4. XML External Entities (XXE)
5. Control de acceso roto
6. Configuración insegura
7. Cross-Site Scripting (XSS)
8. Deserialización insegura
9. Uso de componentes con vulnerabilidades conocidas
10. Insuficiente registro y monitoreo

OWASP Top Ten

Checklist de Seguridad para Aplicaciones Web

1. Validación y Saneamiento de Entradas

- ¿Se validan todas las entradas del usuario en el servidor?
- ¿Se aplican whitelist de valores permitidos siempre que sea posible?
- ¿Se utiliza encoding/escaping apropiado para HTML, XML, JSON, comandos del sistema y consultas SQL?

2. Autenticación y Gestión de Sesiones

- ¿Se fuerza el uso de contraseñas fuertes (mín. 8–12 caracteres, mezcla de mayúsculas, minúsculas, números y símbolos)?
- ¿Se implementa bloqueo de cuenta tras varios intentos fallidos?
- ¿Se utiliza siempre transporte cifrado (HTTPS) para envío de credenciales y tokens?
- ¿Los tokens de sesión son largos, aleatorios y se regeneran tras login o cambio de privilegios?
- ¿Se invalidan las sesiones en el servidor al hacer logout o tras inactividad?

3. Control de Acceso / Autorización

- ¿Cada operación crítica verifica en backend el rol/permiso del usuario?
- ¿No se exponen datos de endpoints o campos que el usuario no debería ver?
- ¿Se aplica el principio de menor privilegio (“least privilege”) en todos los componentes?

4. Criptografía

- ¿Se cifran datos sensibles en reposo (bases de datos, ficheros)?
- ¿Se utilizan algoritmos modernos (AES-256, RSA-2048/3072, ECC) y bibliotecas actualizadas?
- ¿Las claves privadas nunca están “hardcoded” en el código ni en repositorios?
- ¿Se gestionan correctamente los certificados TLS y se renuevan antes de expirar?

5. Manejo de Errores y Registro (Logging)

- ¿Los mensajes de error no exponen información interna (stack traces, rutas de ficheros, detalles de DB)?
- ¿Se registra en log los eventos de seguridad (login, logout, cambios de permiso, errores críticos)?
- ¿Los logs son inaccesibles para usuarios no autorizados y se protegen contra manipulación?

OWASP Top Ten

6. Dependencias y Componentes de Terceros

- ¿Se revisa periódicamente la lista de dependencias para detectar vulnerabilidades conocidas?
- ¿Se actualizan las librerías y frameworks a versiones seguras?
- ¿Se eliminan componentes innecesarios o sin mantenimiento activo?

7. Configuración Segura

- ¿Los entornos de producción están separados de los de desarrollo y pruebas?
- ¿Se desactivan todos los endpoints, puertos y servicios no usados (ej. paneles de administración, debug)?
- ¿Se aplican políticas CORS, CSP y cabeceras de seguridad (HSTS, X-Frame-Options, X-XSS-Protection)?

Application Security Verification Standard (ASVS)

Es un marco de trabajo y estándar diseñado para evaluar y mejorar la seguridad de las aplicaciones. Proporciona una lista estructurada y detallada de requisitos de seguridad, que pueden ser utilizados como guía para diseñar, desarrollar, probar y auditar aplicaciones seguras.

Objetivos Principales del ASVS

1. **Estandarizar la seguridad:** Crear un lenguaje común para discutir y evaluar la seguridad de las aplicaciones.

2. **Evaluación de seguridad:** Proporcionar requisitos claros para auditar aplicaciones en diferentes niveles de profundidad.

3. **Fomentar el desarrollo seguro:** Ayudar a los equipos de desarrollo a implementar prácticas seguras desde las primeras etapas del ciclo de vida del software.

4. **Flexibilidad:** Adaptarse a diferentes tipos de aplicaciones, desde simples hasta complejas, y a distintas industrias.

Niveles de Verificación en ASVS

1. **Nivel 1 – Oportunidades generales de mejora en seguridad:**

• Básico y adecuado para aplicaciones de bajo riesgo.

• Las pruebas pueden realizarse manualmente o con herramientas automatizadas.

2. **Nivel 2 – Aplicaciones comerciales estándar:**

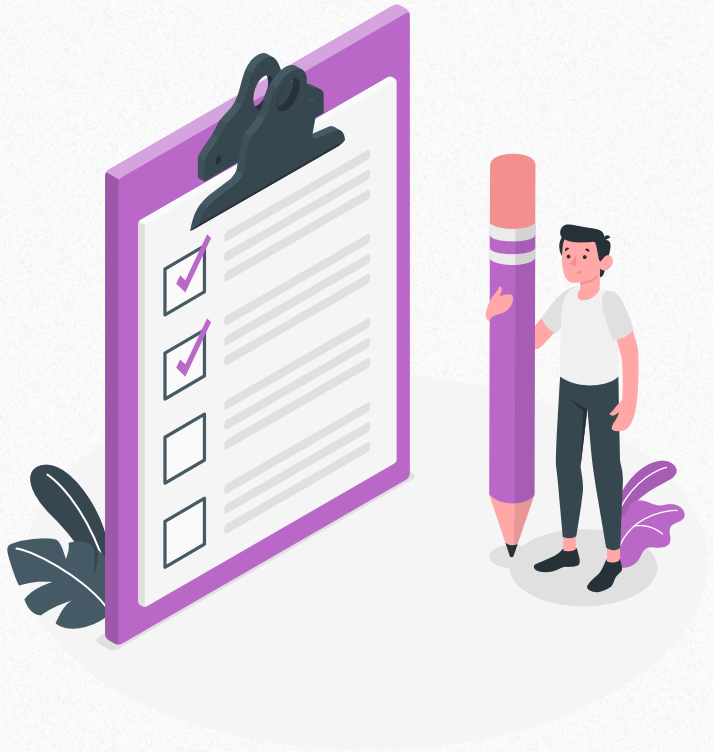
• Requiere un enfoque más riguroso con pruebas manuales y automatizadas.

• Ideal para aplicaciones que manejan datos sensibles o requieren controles adicionales.

3. **Nivel 3 – Aplicaciones de alto riesgo:**

• Diseñado para aplicaciones críticas que requieren la máxima seguridad.

• Implica revisiones exhaustivas y una implementación cuidadosa de los controles de seguridad.



Verificación en ASVS

CODIGO powered by Tecsup/>

1. Autenticación (V2)

Requisito	Descripción	Cumple (Sí/No)
V2.1	¿Los usuarios deben autenticarse antes de acceder a recursos protegidos?	Sí/No
V2.3	¿Las contraseñas cumplen con un mínimo de 12 caracteres y reglas de complejidad?	Sí/No
V2.8	¿El sistema bloquea cuentas tras varios intentos fallidos de autenticación?	Sí/No

2. Gestión de Sesiones (V3)

Requisito	Descripción	Cumple (Sí/No)
V3.2	¿Se invalidan las sesiones activas al cerrar la sesión o cambiar credenciales?	Sí/No
V3.4	¿Los tokens de sesión están protegidos contra exposición en la URL?	Sí/No
V3.7	¿Las sesiones expiran automáticamente después de un período de inactividad?	Sí/No

3. Validación de Entradas (V5)

Requisito	Descripción	Cumple (Sí/No)
V5.1	¿Se validan todas las entradas de usuario contra un conjunto estricto de reglas?	Sí/No
V5.5	¿El sistema utiliza mecanismos para prevenir inyecciones SQL?	Sí/No
V5.8	¿Se rechazan caracteres o datos inesperados en los campos de entrada?	Sí/No

Verificación en ASVS

CODIGO powered by Tecsup/>

4. Gestión de Criptografía (V7)

Requisito	Descripción	Cumple (Sí/No)
V7.2	¿Se usan algoritmos seguros, como AES con una clave de al menos 256 bits?	Sí/No
V7.4	¿Las contraseñas se almacenan con un algoritmo de hashing fuerte (p. ej., bcrypt o Argon2)?	Sí/No
V7.7	¿Se verifica la integridad de los datos cifrados antes de procesarlos?	Sí/No

5. Seguridad de la API (V13)

Requisito	Descripción	Cumple (Sí/No)
V13.1	¿Todas las solicitudes a la API están autenticadas?	Sí/No
V13.3	¿Los datos sensibles enviados a través de la API están cifrados en tránsito (HTTPS)?	Sí/No
V13.7	¿La API tiene límites de tasa (rate limiting) para prevenir abusos?	Sí/No

6. Registro y Monitoreo (V10)

Requisito	Descripción	Cumple (Sí/No)
V10.1	¿Los eventos críticos (autenticación, cambios de configuración, errores) se registran?	Sí/No
V10.3	¿Los registros no contienen información sensible como contraseñas o tokens de sesión?	Sí/No
V10.5	¿El sistema detecta y notifica accesos no autorizados o intentos de ataque?	Sí/No

Thread Modeling: Diseñando para la Seguridad

“Pensar como un atacante para proteger lo que construyes.”

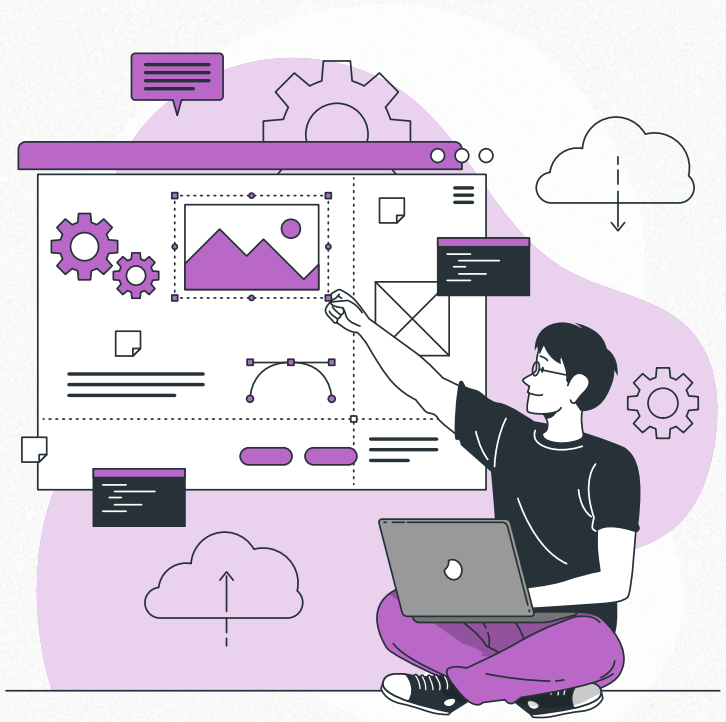
¿Qué es el Modelado de Amenazas?

El modelado de amenazas es un **proceso estructurado** que permite a los equipos de desarrollo identificar y evaluar las amenazas potenciales en un sistema.

Su objetivo es **anticiparse a los ataques** y diseñar mitigaciones antes de que se implementen los sistemas.

Importancia:

- Ayuda a **priorizar esfuerzos de seguridad**, enfocándose en las amenazas más probables y con mayor impacto.
- Es **proactivo** en lugar de reactivo: identificas problemas antes de que ocurran.
- Reduce **costos y riesgos**: corregir una vulnerabilidad en producción puede costar hasta **10 veces más** que prevenirla en la fase de diseño.



Beneficios del Modelado de Amenazas

1. **Identificación Temprana de Vulnerabilidades:** En lugar de descubrir problemas en pruebas o, peor aún, en producción, puedes diseñar sistemas para ser intrínsecamente seguros.
2. **Mitigación Proactiva:** No solo identificas amenazas, sino que diseñas soluciones para ellas antes de que un atacante las explote.
3. **Comunicación entre Equipos:** Facilita discusiones entre desarrolladores, testers y expertos en seguridad. Es una forma visual y lógica de entender el sistema.
4. **Priorización de Recursos:** Al conocer los riesgos más críticos, los equipos pueden enfocarse en mitigar las amenazas con mayor impacto o probabilidad.

Componentes Clave del Modelado de Amenazas

1. **Activo:** Cualquier recurso que quieras proteger.
Ejemplo: Datos personales, contraseñas, tokens de autenticación.

2. **Amenaza:** Un evento o acción que podría comprometer un activo.
Ejemplo: Inyección SQL en un formulario.

3. **Atacante:** El actor que intenta explotar las vulnerabilidades.
Ejemplo: Un hacker malicioso, un empleado interno deshonesto.

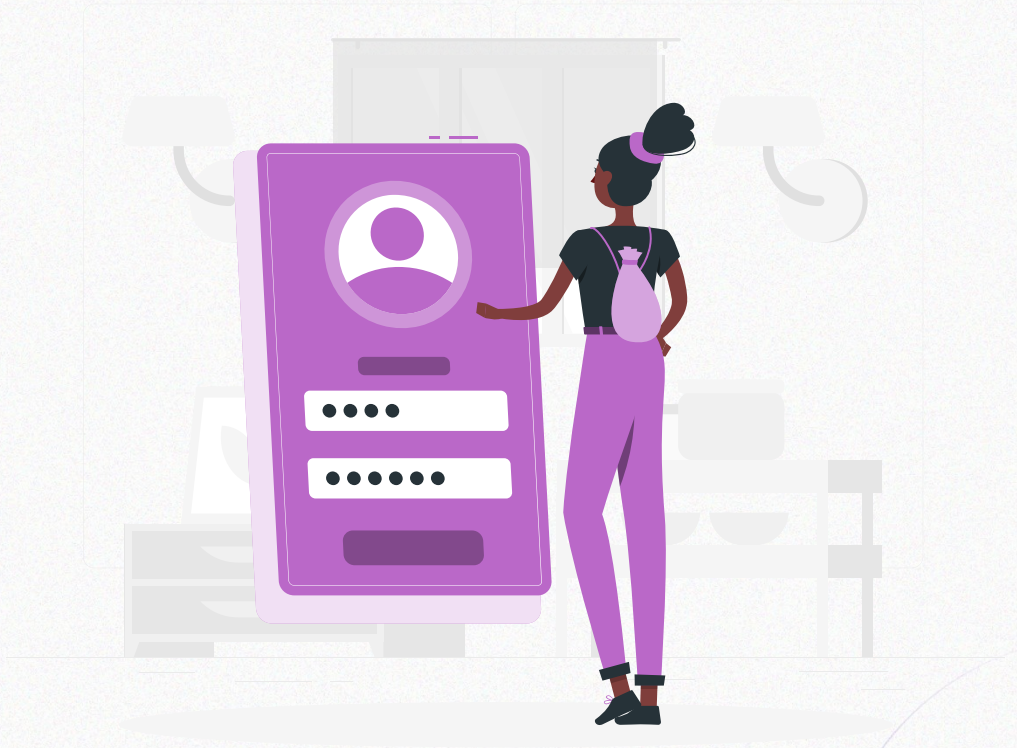
4. **Vulnerabilidad:** Una debilidad en el sistema que el atacante puede explotar.
Ejemplo: Una API que no valida entradas del usuario.

5. **Mitigación:** Medidas que implementas para reducir el impacto o la probabilidad de la amenaza.
Ejemplo: Uso de un WAF para prevenir inyecciones SQL.

Ejemplo Práctico: Pensando como un Atacante

Escenario: Una API de autenticación

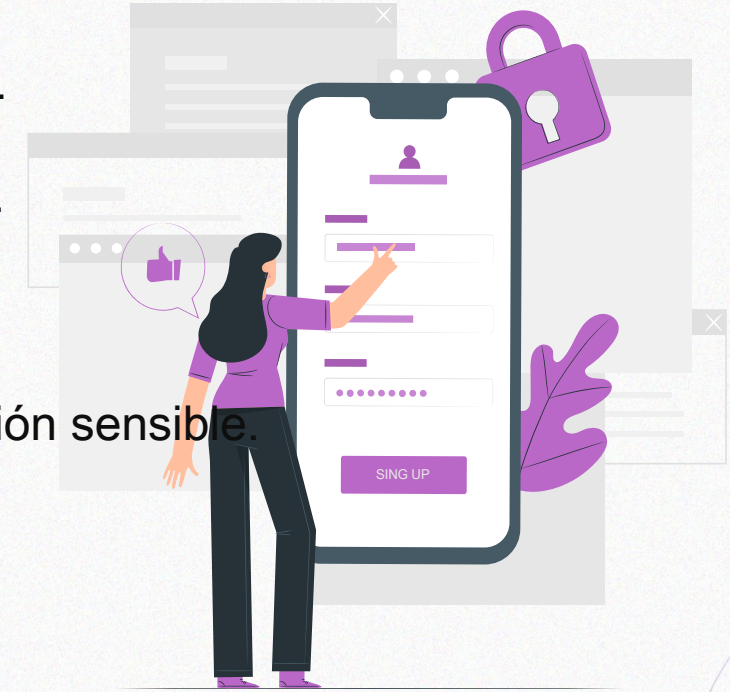
- **Activo:** Credenciales de usuario.
- **Amenaza:** Un atacante podría intentar un ataque de fuerza bruta para adivinar contraseñas.
- **Vulnerabilidad:** La API no tiene un límite de intentos de inicio de sesión.
- **Mitigación:** Implementar bloqueo temporal después de 5 intentos fallidos.



STRIDE

Desarrollado por Microsoft, STRIDE es un modelo que categoriza amenazas en seis áreas:

- **S**poofing (Suplantación de identidad): Falsificar credenciales.
- **T**ampering (Alteración): Modificación no autorizada de datos.
- **R**epudiation (Negación): Incapacidad de rastrear acciones.
- **I**nformation Disclosure (Divulgación): Exposición de información sensible.
- **D**enial of Service (DoS): Interrupción del servicio.
- **E**levation of Privilege (EoP): Ganar acceso indebido.



LINDDUN

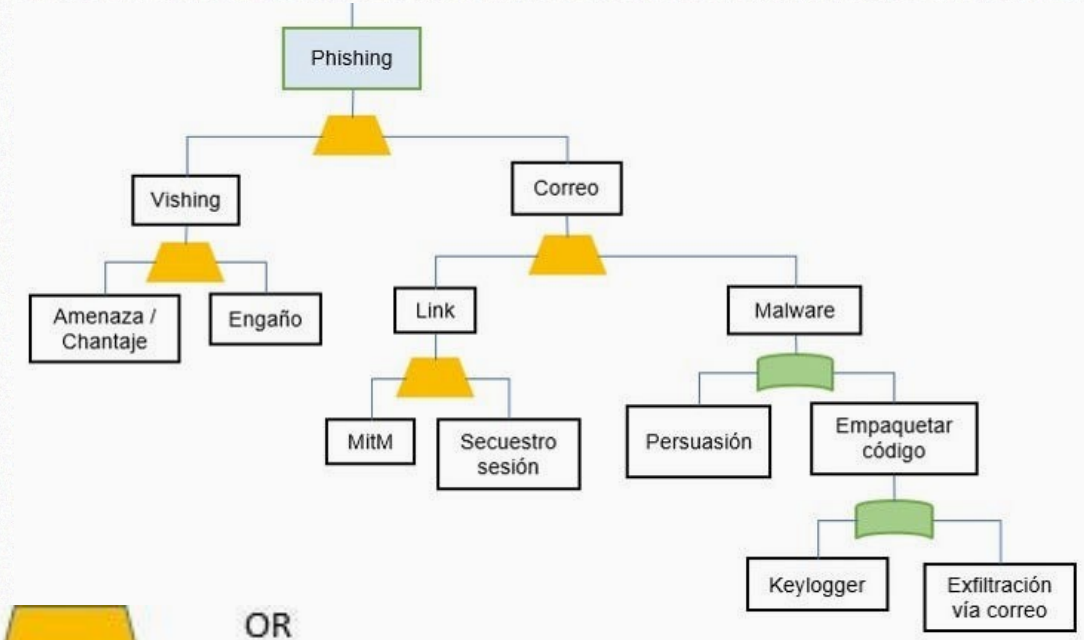
Diseñado para abordar **amenazas de privacidad**, LINDDUN es ideal para sistemas que manejan datos sensibles:

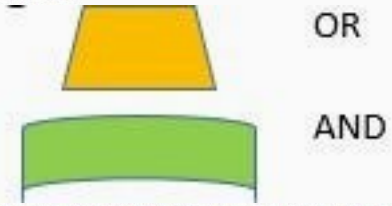
- **L**inkability: Asociar datos entre sistemas.
- **I**dentifiability: Identificar usuarios de forma no autorizada.
- **N**on-repudiation: Incapacidad de negar una acción.
- **D**etectability: Descubrir datos sensibles o su existencia.
- **D**isclosure of Information: Divulgación de datos no autorizados.
- **U**nwanted Policy Enforcement: Aplicación indebida de políticas.

Attack Trees

Representa gráficamente cómo un atacante puede alcanzar su objetivo:

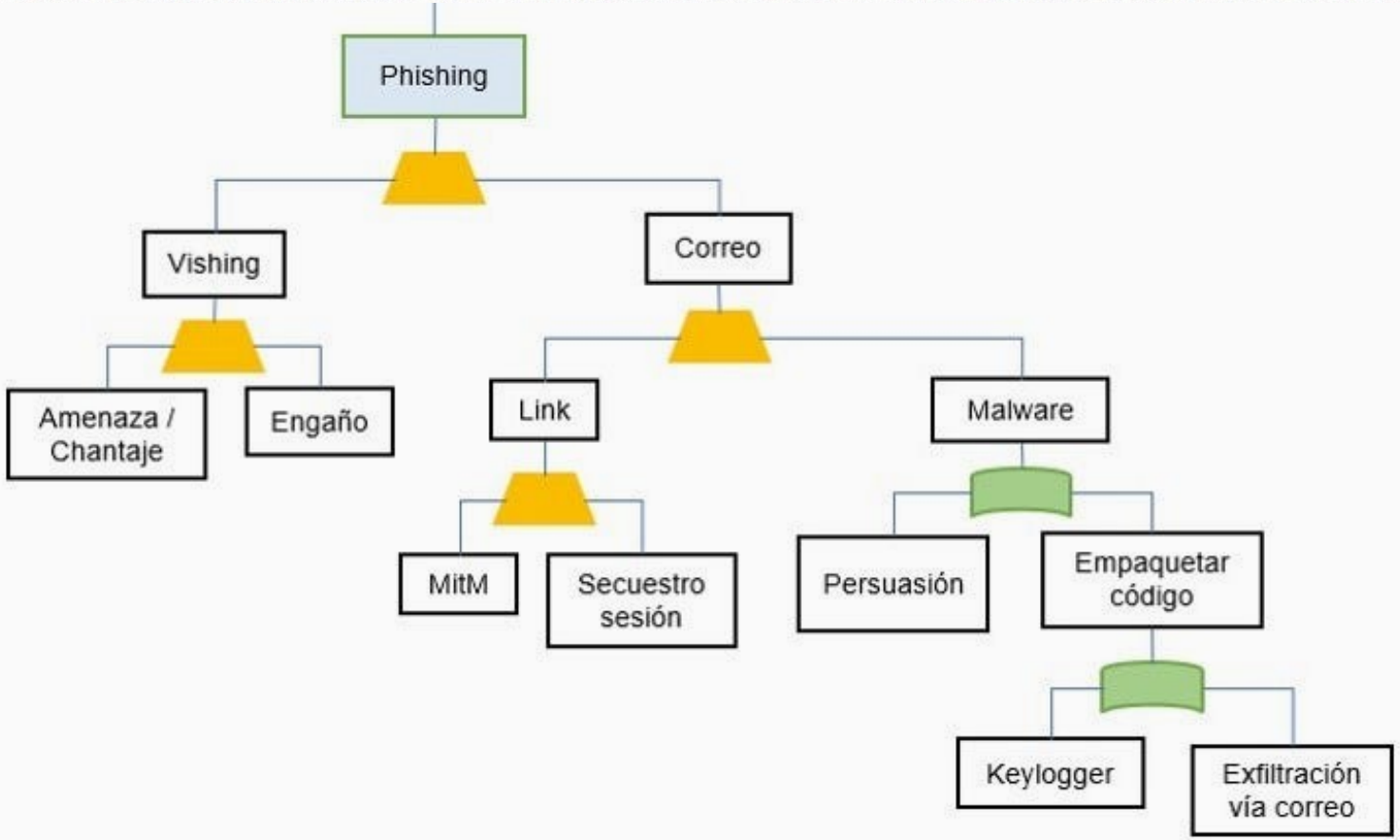
- **Raíz:** El objetivo del atacante (por ejemplo, robar datos).
- **Ramas:** Métodos posibles para lograr el objetivo.
- **Relaciones:** Muestran cómo se conectan las acciones entre ellas
  - **Relación AND:** es posible que para realizar la siguiente acción se deban cumplir una serie de acciones previas.
  - **Relación OR:** también es posible, que se pueda pasar a la siguiente acción si solo se cumple una acción anterior.





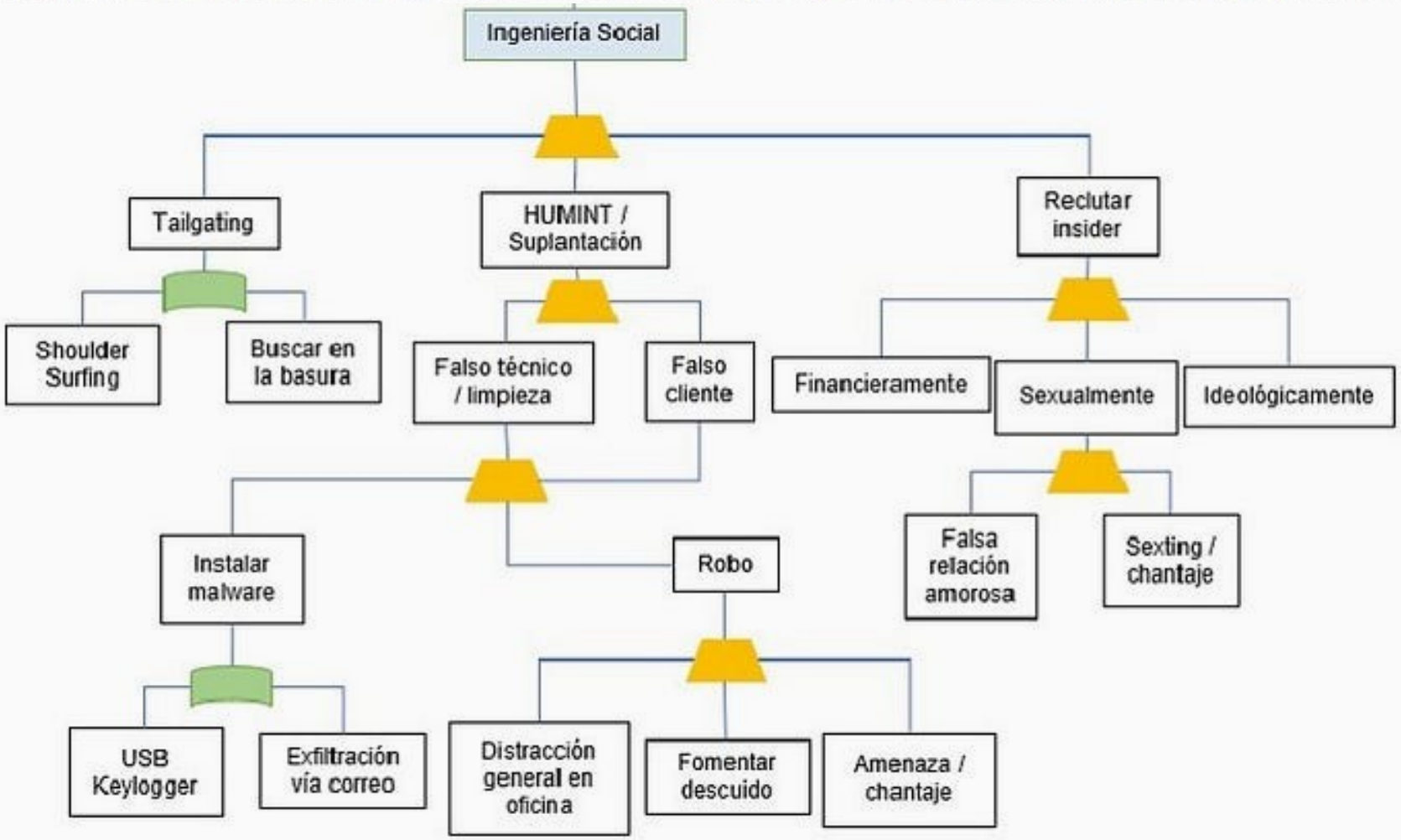
Attack Trees

Representa gráficamente cómo un atacante puede alcanzar su objetivo:



Attack Trees

Representa gráficamente cómo un atacante puede alcanzar su objetivo:



Identificación y Priorización de Riesgos

1. **Cómo identificar riesgos:**
   - Utilizar diagramas de flujo de datos (DFD) para visualizar flujos y puntos de entrada.
   - Revisar cada flujo en busca de amenazas STRIDE o LINDDUN.
2. **Priorizar riesgos según:**
   - **Impacto:** ¿Qué tan grave sería el daño?
   - **Probabilidad:** ¿Qué tan probable es que ocurra?
3. **Matriz de Riesgos:**
   - Crear una tabla donde se clasifiquen las amenazas según impacto y probabilidad.

¿Qué es el Threat Modeling?

Es un proceso estructurado para identificar, evaluar y priorizar amenazas potenciales en un sistema, considerando su arquitectura, diseño, implementación y operaciones. El objetivo principal es **entender cómo un atacante podría comprometer el sistema y qué medidas tomar para prevenirlo**.

Objetivos del Threat Modeling

1. **Identificar amenazas específicas:** Entender los riesgos asociados al sistema.

2. **Evaluar el impacto:** Analizar cómo esas amenazas podrían afectar la confidencialidad, integridad o disponibilidad.

3. **Priorizar los riesgos:** Determinar cuáles amenazas deben mitigarse primero según su impacto y probabilidad.

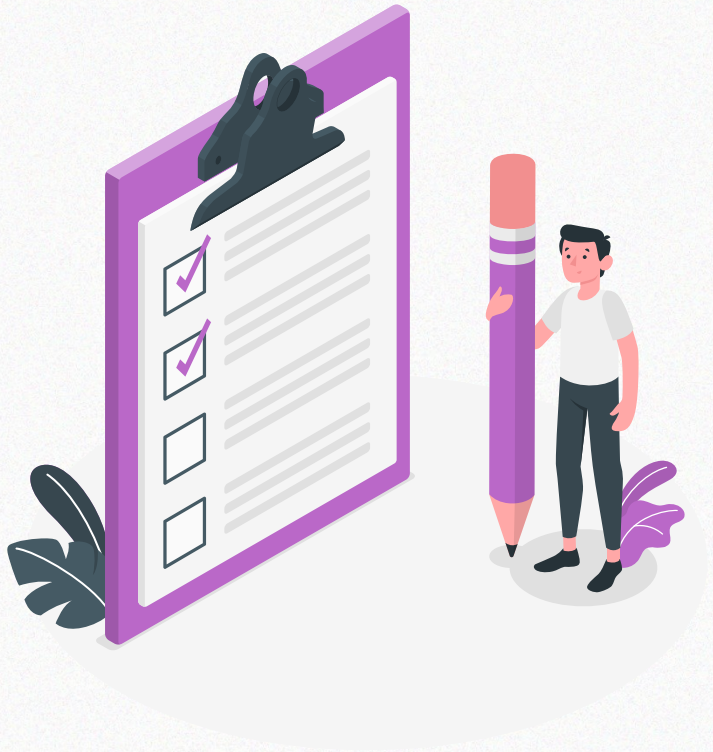
4. **Diseñar controles de seguridad:** Implementar contramedidas efectivas para proteger el sistema.

Threat Modeling es un proceso estructurado para identificar y priorizar amenazas en un sistema, con el fin de definir controles que mitiguen los riesgos antes de que se conviertan en vulnerabilidades.

Proceso del Threat Modeling

Conceptos

- **Activo**: Cualquier recurso del sistema que tenga valor para el negocio (ej. datos personales, API de pagos).
- **Amenaza**: Potencial ataque o evento que comprometa un activo.
- **Vulnerabilidad**: Debilidad que puede ser explotada.
- **Control**: Medida que mitiga una amenaza.



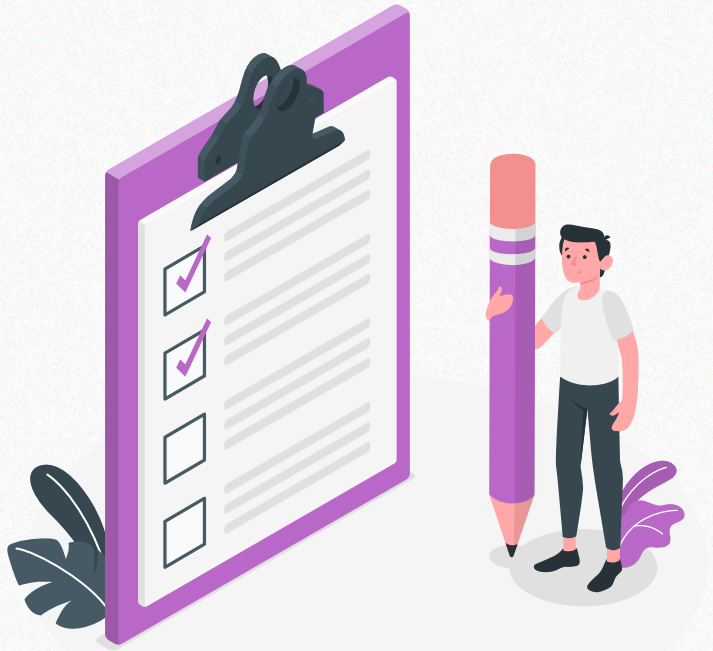
"Activos atraen amenazas, vulnerabilidades abren puertas, controles las cierran."

Proceso del Threat Modeling

Paso 1: Entender el sistema

Antes de identificar amenazas, es esencial comprender el sistema y sus componentes:

- **Arquitectura:** Diagramas de alto nivel que muestran componentes, conexiones y flujos de datos.
- **Activos críticos:** Datos sensibles, servicios esenciales o cualquier recurso que deba protegerse.
- **Contexto:** Identificar usuarios, actores externos e interacciones relevantes.



Herramientas útiles:

• Diagramas de flujo de datos (DFD).

• Diagramas de arquitectura (por ejemplo, diagrama C4).

Sistema: Aplicación de pedidos de comida

Componentes:
- Frontend móvil (Android/iOS)
- API Gateway
- Backend de pedidos
- Base de datos (DB)
- Servicio de pagos externo (Stripe/PayPal)

Actores:
- Usuario final
- Repartidor
- Administrador del restaurante

Activos críticos:
- Datos de tarjetas de crédito
- Credenciales de usuarios
- Información de pedidos

Proceso del Threat Modeling

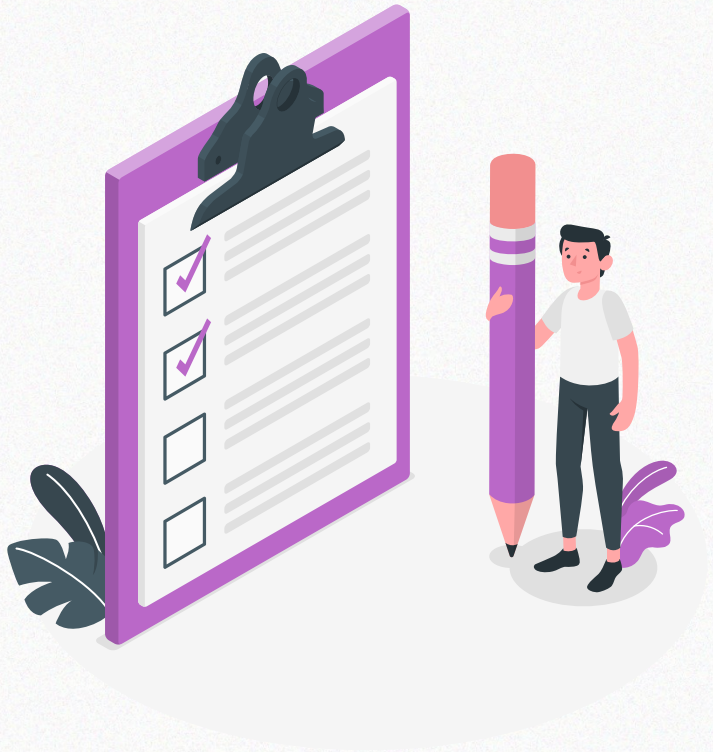
Paso 2: Identificar amenazas

Existen varias técnicas para identificar amenazas, como:

Modelos basados en taxonomías:

Usar marcos como **STRIDE**, que clasifica amenazas según:

- **S**poofing (suplantación).
- **T**ampering (alteración de datos).
- **R**epudiation (repudio de acciones).
- **I**nformation disclosure (divulgación de información).
- **D**enial of Service (denegación de servicio).
- **E**levation of Privilege (escalamiento de privilegios).



Identificar amenazas

¿Qué podría salir mal en cada componente y flujo de datos del sistema?

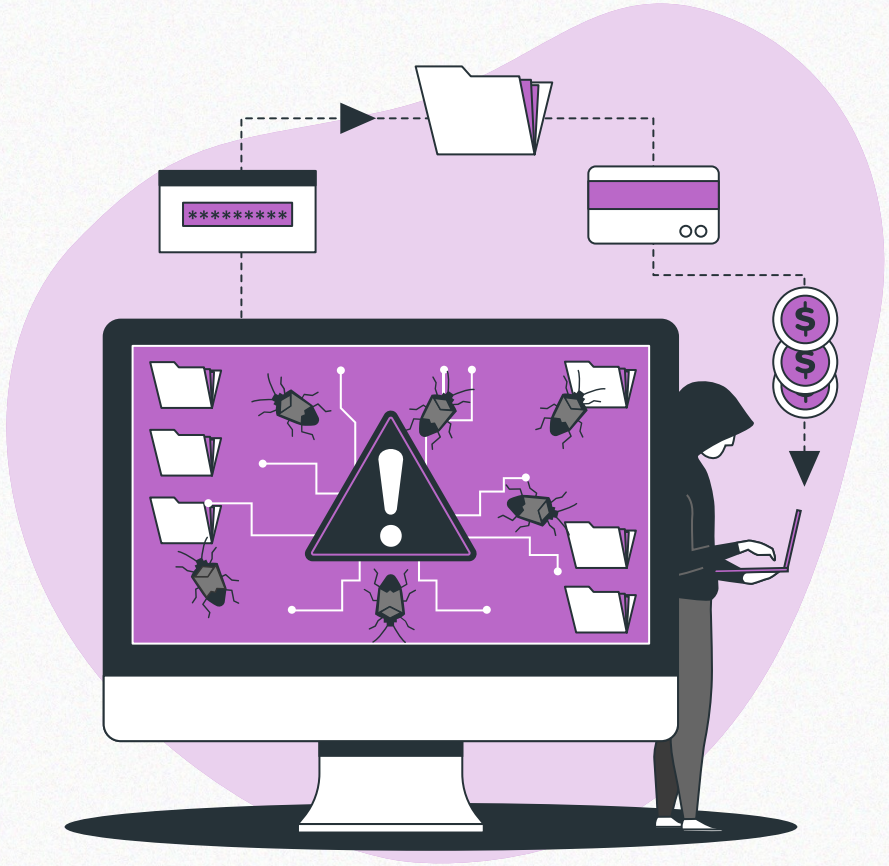
STRIDE	Ejemplo en la app
S – Spoofing	Un atacante se hace pasar por un usuario usando credenciales robadas o tokens falsificados.
T – Tampering	Modificación del saldo en una petición API interceptada.
R – Repudiation	Usuario niega haber realizado una transferencia.
I – Information Disclosure	Un endpoint expone datos personales de otros clientes por un fallo en autorización.
D – Denial of Service	Bot inunda el servidor con millones de peticiones de login, dejándolo fuera de servicio.
E – Elevation of Privilege	Un usuario común explota un bug y obtiene acceso al panel de administración.

- ¿Qué harías si fueras un hacker con acceso a la red?
- ¿Qué pasa si alguien modifica los datos en tránsito entre frontend y backend?
- ¿El sistema confía demasiado en la información enviada por el cliente?
- ¿Podría alguien obtener más permisos de los que debería?
- ¿Qué pasaría si un atacante envía 1000 peticiones por segundo?

Herramientas útiles para este paso

¿Qué podría salir mal en cada componente y flujo de datos del sistema?

- **OWASP Cheat Sheets**: Guías rápidas para cada categoría de amenaza.
- **Threat Dragon (OWASP)**: Herramienta para dibujar amenazas en diagramas.
- **Microsoft Threat Modeling Tool**: Integra STRIDE en el DFD automáticamente.



Proceso del Threat Modeling

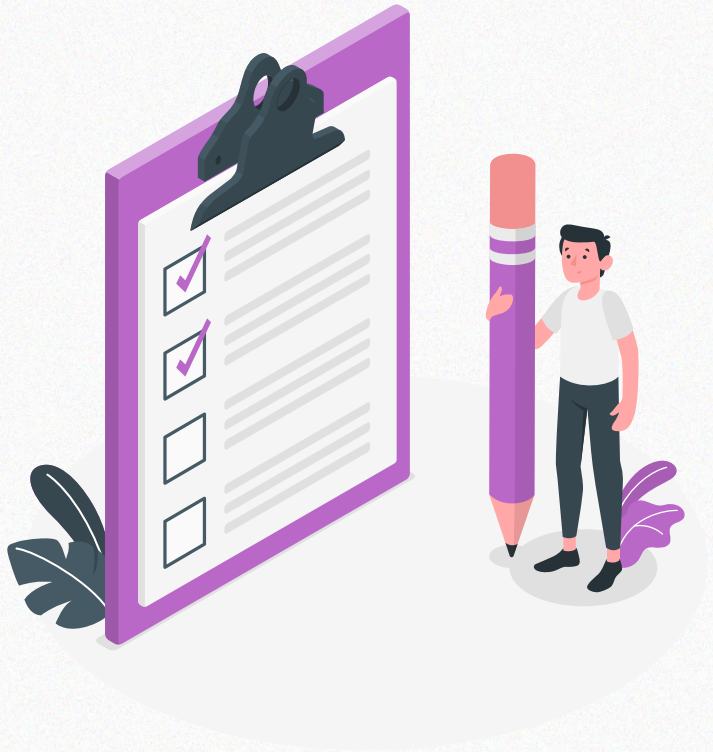
Paso 3: Evaluar y priorizar las amenazas

Cada amenaza debe ser evaluada en términos de:

- **Probabilidad:** ¿Qué tan probable es que ocurra?
- **Impacto:** ¿Cuál sería el efecto si ocurriera?

Ejemplo de evaluación:
Utilizar un enfoque como el **DREAD**, que evalúa amenazas en función de:

- **D**amage (daño).
- **R**eproducibility (capacidad de reproducirse).
- **E**xploitability (facilidad de explotación).
- **A**ffected users (usuarios afectados).
- **D**iscoverability (facilidad de descubrimiento).



Proceso del Threat Modeling

	Baja Probabilidad	Media Probabilidad	Alta Probabilidad
Alto Impacto	🔴 Alta prioridad	🔴 Alta prioridad	🔴 Crítica
Medio Impacto	🟠 Media prioridad	🟠 Media prioridad	🔴 Alta prioridad
Bajo Impacto	🟢 Baja prioridad	🟢 Baja prioridad	🟠 Media prioridad

Evaluación simple (Impacto vs Probabilidad)

- **Probabilidad (Likelihood):** ¿Qué tan probable es que un atacante pueda explotarla?
- **Impacto (Impact):** ¿Qué tan graves serían las consecuencias si se materializa?

Enfoque DREAD (Microsoft)

DREAD es un modelo creado por Microsoft para **evaluar amenazas de forma cuantitativa y priorizarlas** según su riesgo.

actor	Pregunta guía
Damage	¿Cuánto daño causa si se explota?
Reproducibility	¿Es fácil reproducir el ataque?
Exploitability	¿Qué tan fácil es ejecutar el ataque?
Affected Users	¿A cuántos usuarios afecta?
Discoverability	¿Es fácil descubrir la vulnerabilidad?

Amenaza: *Un atacante puede interceptar credenciales porque la API no usa HTTPS.*

DREAD Factor	Evaluación	Puntaje (1-10)
Damage	Robo de datos de usuario, acceso a cuentas bancarias	9
Reproducibility	Cualquier atacante en la red puede reproducirlo	10
Exploitability	Simple, basta un proxy o sniffer de red	9
Affected Users	Todos los usuarios de la app móvil	10
Discoverability	Fácil: tráfico no cifrado visible con herramientas básicas	10

$$\text{Riesgo promedio} = \frac{9 + 10 + 9 + 10 + 10}{5} = 9.6$$

Proceso del Threat Modeling

Paso 4: Diseñar contramedidas

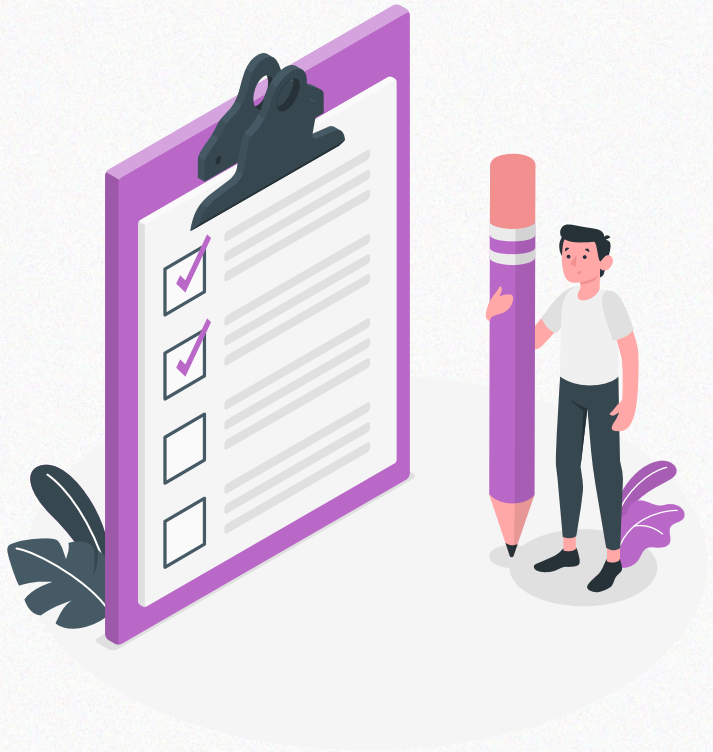
Para cada amenaza priorizada:

1. **Mitigar:** Reducir el impacto o la probabilidad de que ocurra.

2. **Evitar:** Cambiar el diseño del sistema para eliminar la amenaza.

3. **Detectar:** Implementar mecanismos para identificar cuándo ocurre.

4. **Responder:** Diseñar planes de contingencia para limitar el daño.

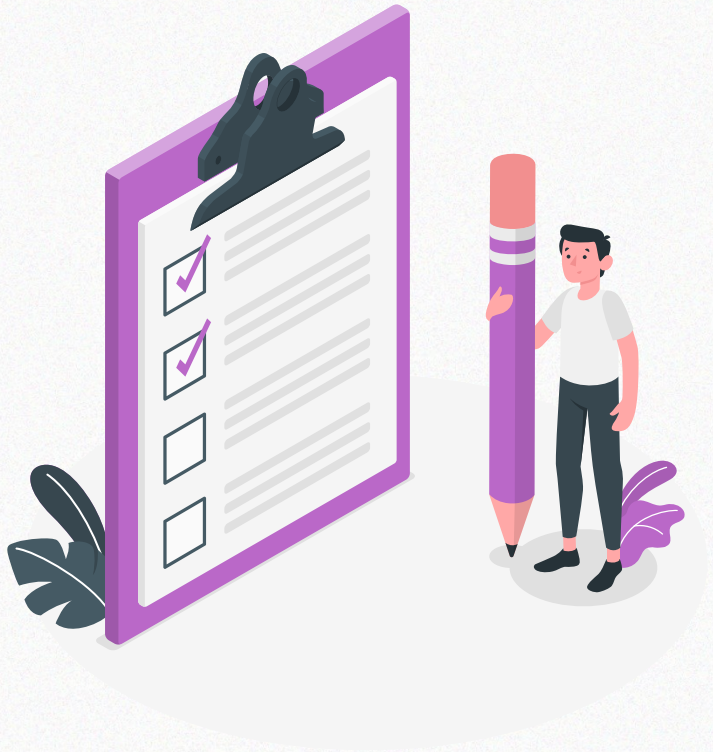


Ejemplo práctico:

- **Amenaza:** Divulgación de datos sensibles en tránsito.
- **Mitigación:** Implementar cifrado de extremo a extremo con TLS.

Proceso del Threat Modeling

- ¿Qué factor de DREAD creen que es más crítico en esta amenaza?
- ¿Un atacante principiante podría explotarlo? ¿O requiere conocimientos avanzados?
- Si tuviéramos que reducir solo **un factor** (por ejemplo, Discoverability), ¿cómo lo haríamos?



Proceso del Threat Modeling

Amenaza: *Un atacante puede interceptar credenciales porque la API no usa HTTPS.*

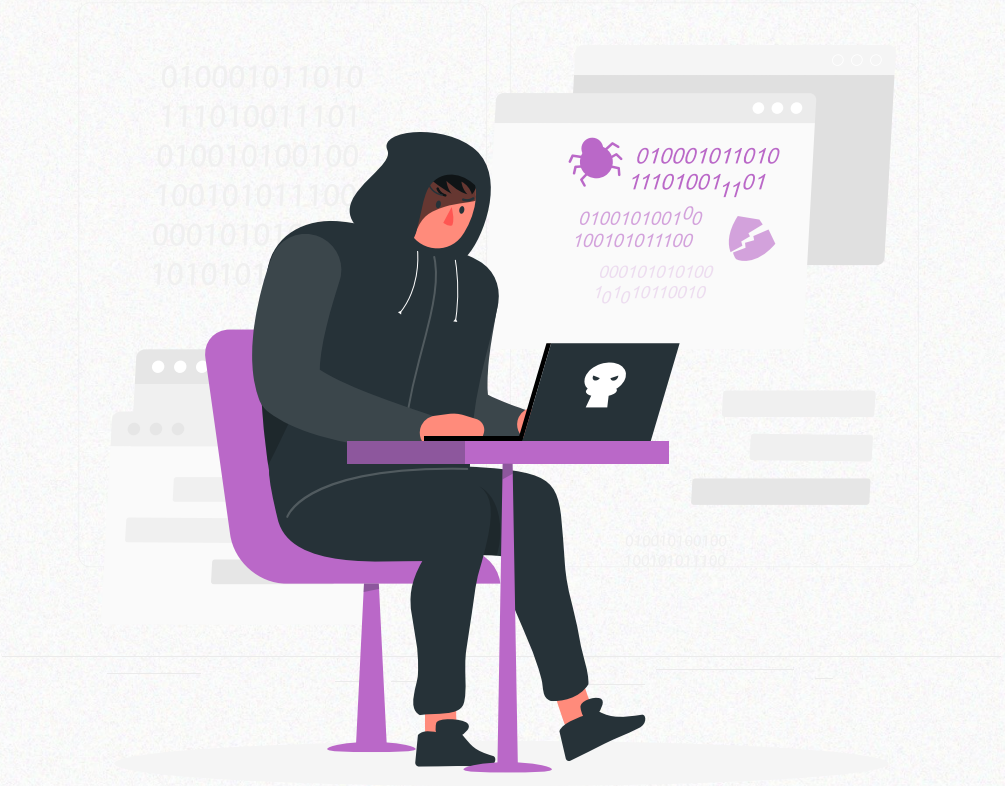
Estrategia	Contramedida
Mitigar	Implementar cifrado de extremo a extremo (TLS 1.3) para todo el tráfico HTTP.
Evitar	Cambiar el diseño para que nunca se transmitan credenciales en texto plano: usar OAuth2 con tokens de acceso temporales.
Detectar	Configurar un IDS (Intrusion Detection System) para alertar sobre tráfico HTTP no cifrado en la red interna.
Responder	Diseñar un playbook: revocar sesiones comprometidas y forzar reseteo de contraseñas en caso de fuga detectada.

Amenazas Comunes

Ataques de denegación de servicio (DoS).

Un ataque DDoS ocurre cuando múltiples dispositivos (generalmente parte de una red de bots o botnet) envían un volumen masivo de tráfico hacia un servidor, aplicación o red, con la intención de:

- **Saturar los recursos** del sistema, como ancho de banda, CPU o memoria.
- **Interrumpir el acceso legítimo** de usuarios al servicio.



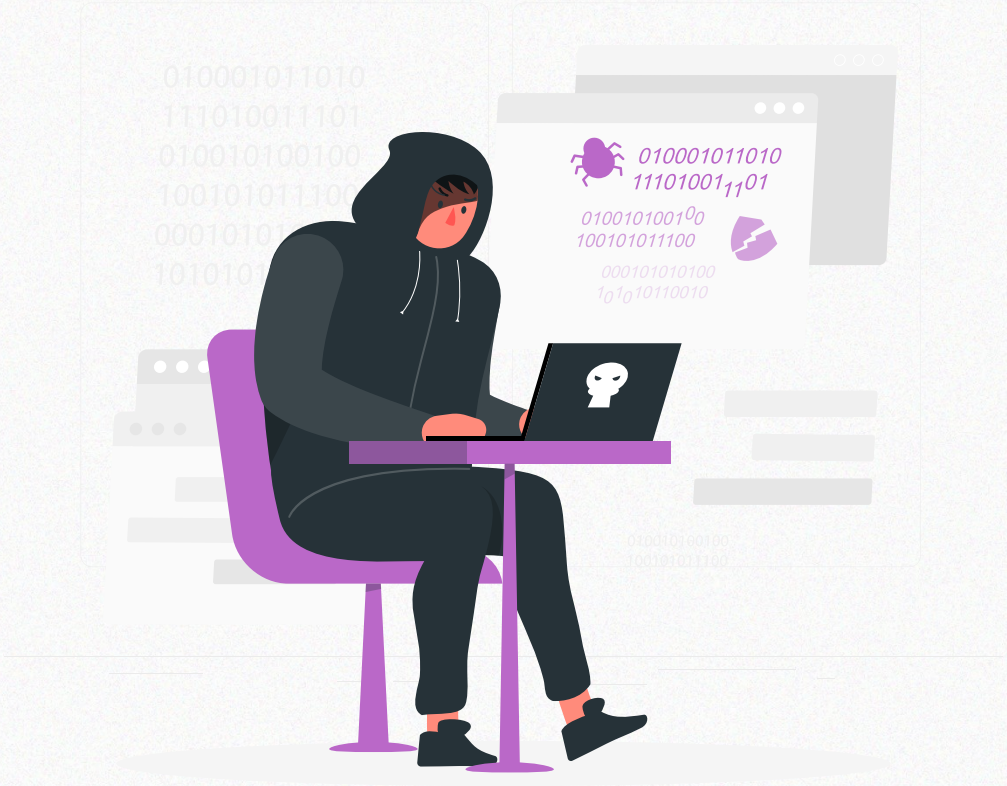
Cómo afecta el diseño arquitectónico

Los DDoS explotan limitaciones en:

1. **Infraestructura**: Como ancho de banda de red o capacidad de procesamiento del servidor.

2. **Diseño de la aplicación**: Funciones que no están optimizadas para manejar muchas solicitudes simultáneamente.

3. **Manejo de tráfico**: Ausencia de mecanismos para distinguir tráfico legítimo del malicioso.

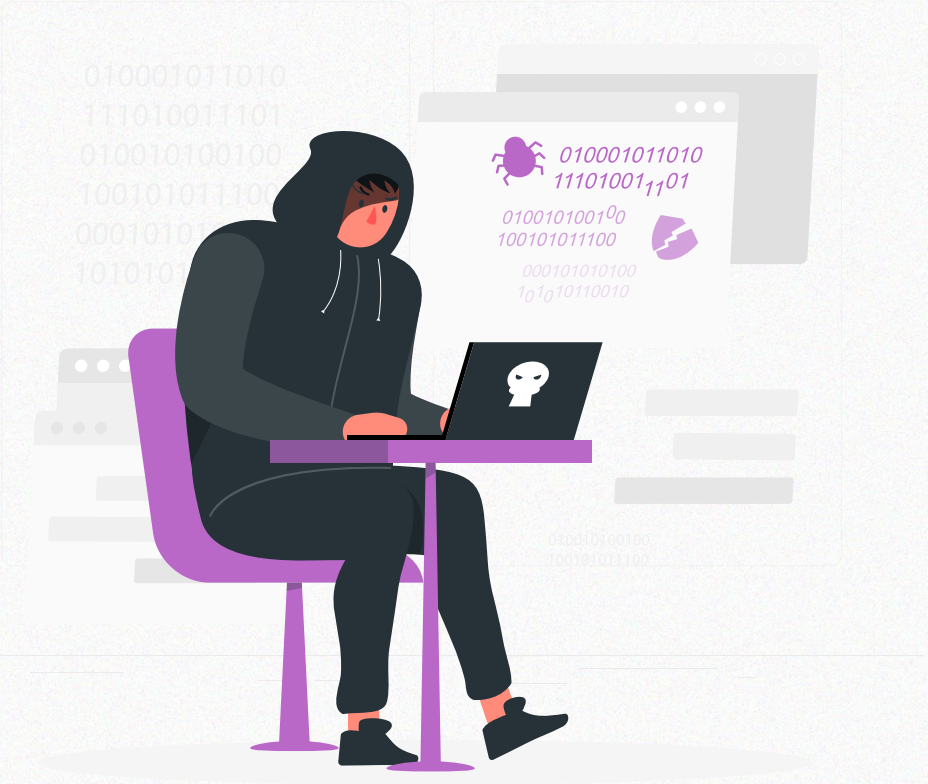


A01:2021 - Broken Access Control

Las reglas que controlan el acceso a recursos no están correctamente implementadas o son fáciles de evadir.

Para evitarlo un arquitecto debería:

- Diseñar y documentar un modelo claro de control de acceso (por roles y permisos).
- Implementar controles de acceso en el servidor (no confiar en validaciones del cliente).
- Usar un framework de seguridad confiable que implemente control de acceso.
- Auditar y probar exhaustivamente las políticas de acceso.
- Minimizar la exposición de endpoints sensibles.

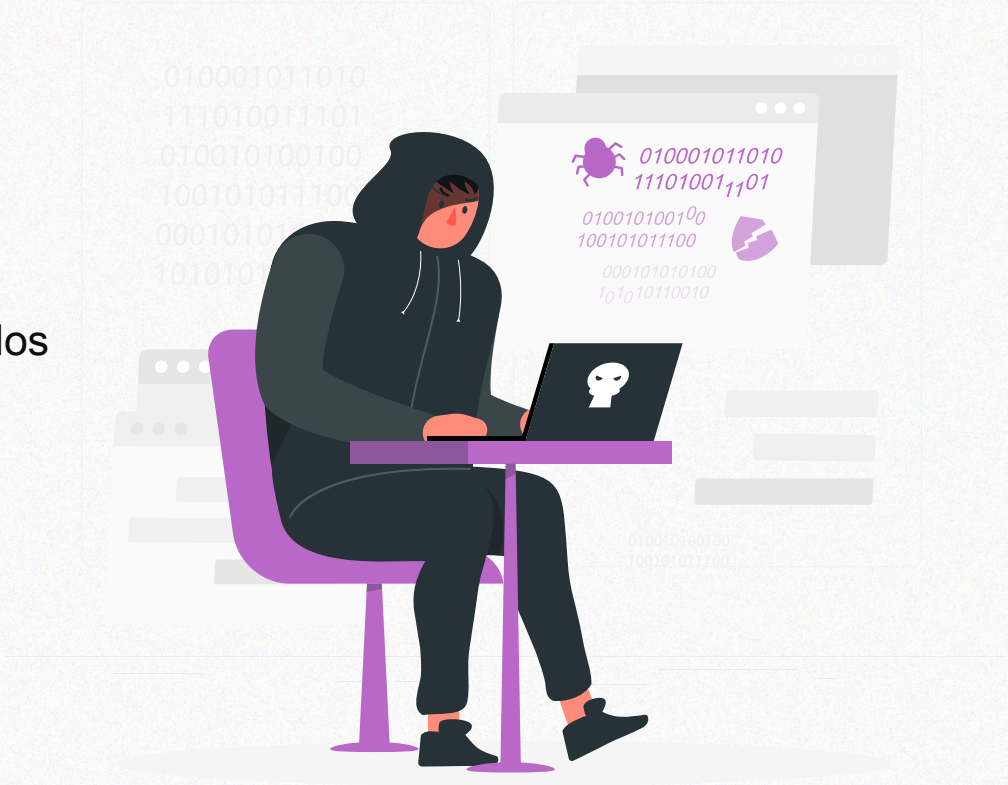


A02:2021 - Cryptographic Failures

Datos sensibles no están protegidos adecuadamente, lo que puede llevar a exposiciones.

Para evitarlo un arquitecto debería:

- Definir estándares criptográficos organizacionales y usarlos consistentemente (e.g., AES-256, RSA).
- Implementar transporte seguro (TLS) para todas las comunicaciones.
- Evitar almacenar datos sensibles, salvo que sea absolutamente necesario.
- Usar bibliotecas de criptografía probadas y evitar implementar algoritmos personalizados.
- Diseñar procesos seguros para el almacenamiento y manejo de claves.

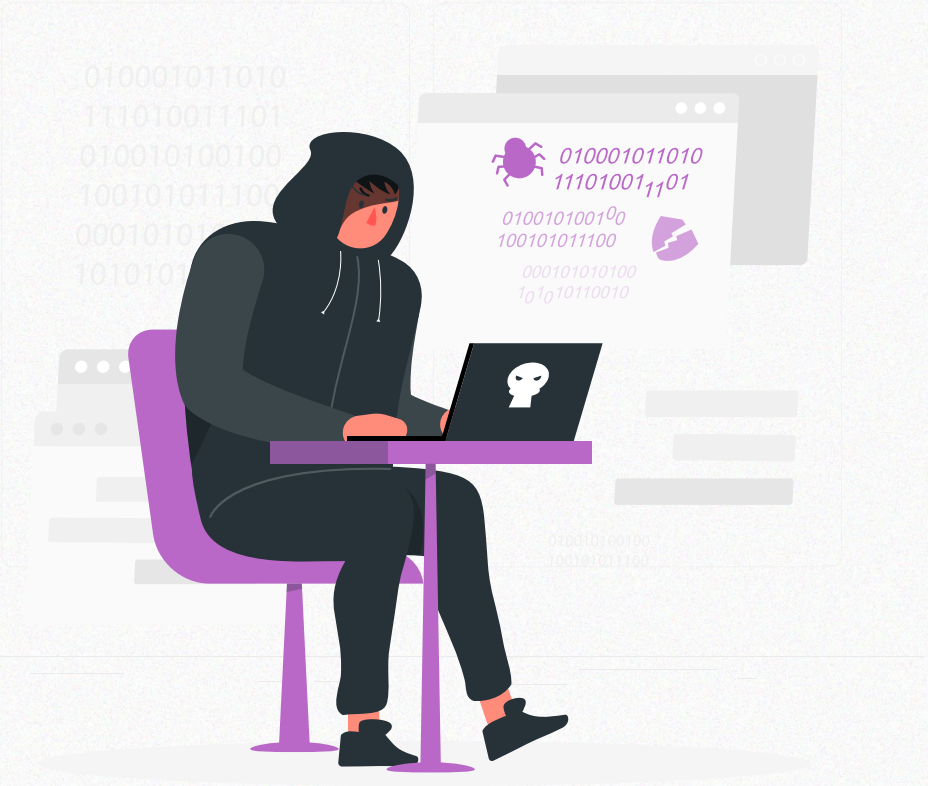


A03:2021 - Injection

Un atacante puede inyectar datos maliciosos (e.g., SQL, NoSQL, LDAP) en las entradas para manipular la lógica del sistema.

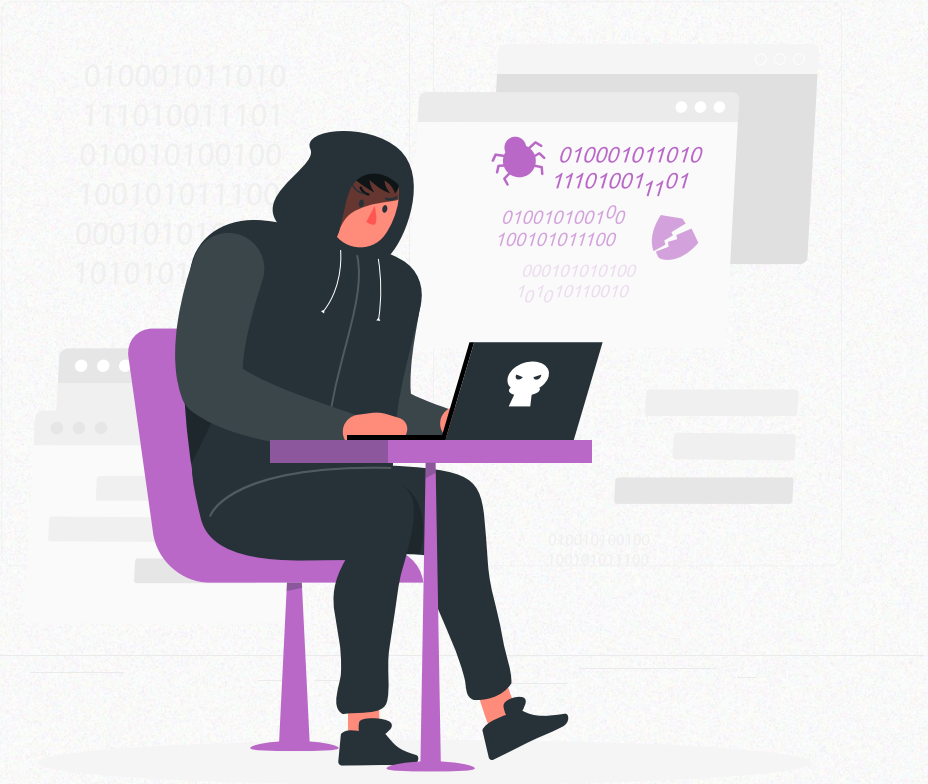
Para evitarlo un arquitecto debería:

- Aplicar principios de validación de entrada estricta.
- Utilizar consultas preparadas (con parámetros) o procedimientos almacenados.
- Implementar sanitización de datos en entradas provenientes de usuarios.
- Diseñar sistemas con el principio de privilegio mínimo en bases de datos.



A04:2021 - Insecure Design

Falta de consideración de prácticas de diseño seguro durante la arquitectura y el desarrollo.



Para evitarlo un arquitecto debería:

- Aplicar modelado de amenazas durante el diseño.
- Definir requisitos de seguridad desde la fase inicial del proyecto.
- Implementar patrones de diseño seguro (e.g., validación centralizada, controles redundantes).
- Integrar revisiones de diseño de seguridad en las etapas iniciales del desarrollo.

A05:2021 - Security Misconfiguration

Configuraciones inseguras en servidores, aplicaciones, frameworks o bases de datos.

Para evitarlo un arquitecto debería:

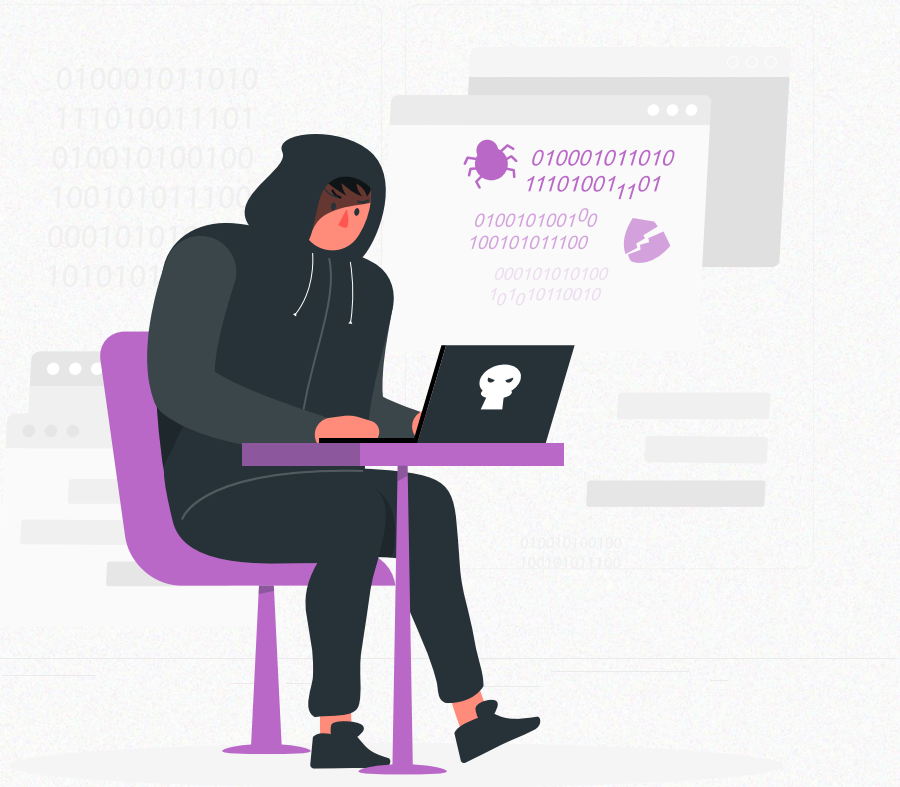
• Diseñar procesos para la gestión de configuración segura (Infrastructure as Code).

• Habilitar únicamente las características y servicios necesarios.

• Mantener el software actualizado y aplicar parches de seguridad.

• Usar escáneres automatizados para detectar configuraciones inseguras.

• Centralizar y controlar configuraciones en un sistema de gestión de cambios.

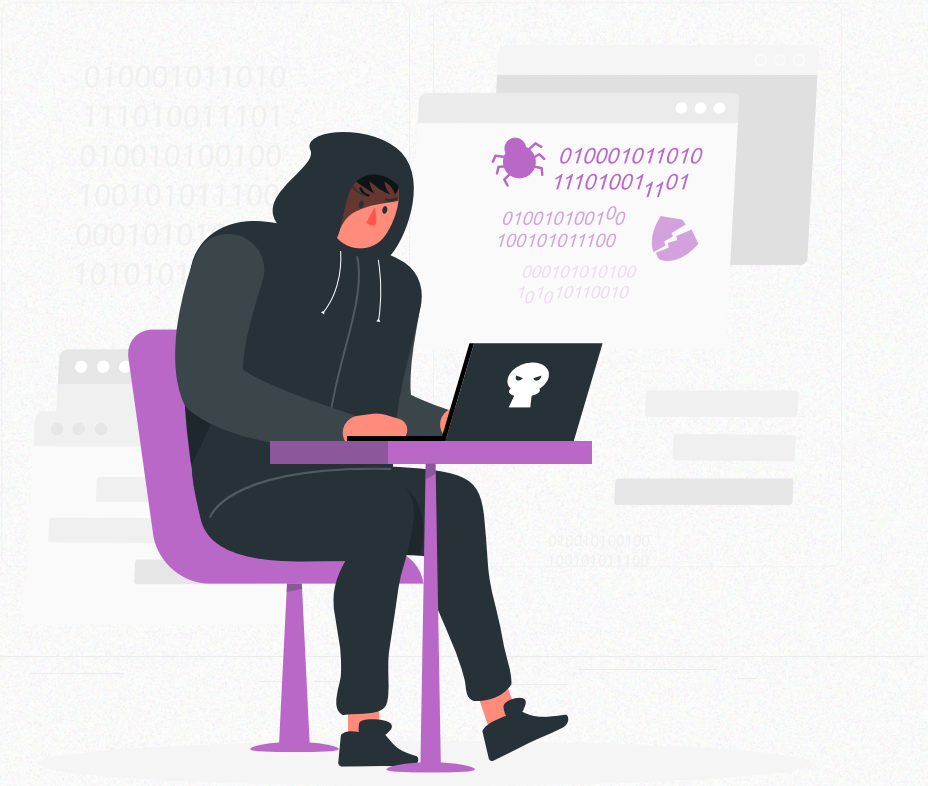


A06:2021 - Vulnerable and Outdated Components

Uso de librerías, frameworks o componentes desactualizados con vulnerabilidades conocidas.

Para evitarlo un arquitecto debería:

- Definir un proceso para mantener y actualizar componentes de software.
- Implementar herramientas de escaneo de dependencias para detectar versiones inseguras.
- Diseñar un proceso de parcheo y actualización continuo.
- Evitar usar componentes que no tengan soporte activo del proveedor.



A07:2021 - Identification and Authentication Failures

Implementación débil de mecanismos de autenticación e identificación.

Para evitarlo un arquitecto debería:

- Diseñar flujos de autenticación robustos usando estándares (e.g., OAuth, SAML).
- Requerir autenticación multifactor para operaciones críticas.
- Implementar controles de bloqueo ante múltiples intentos fallidos.
- Asegurar las contraseñas almacenadas mediante hashing seguro (e.g., bcrypt).
- Evitar exponer información en mensajes de error de autenticación.

A08:2021 - Software and Data Integrity Failures

Problemas de integridad de datos o componentes críticos.

Para evitarlo un arquitecto debería:

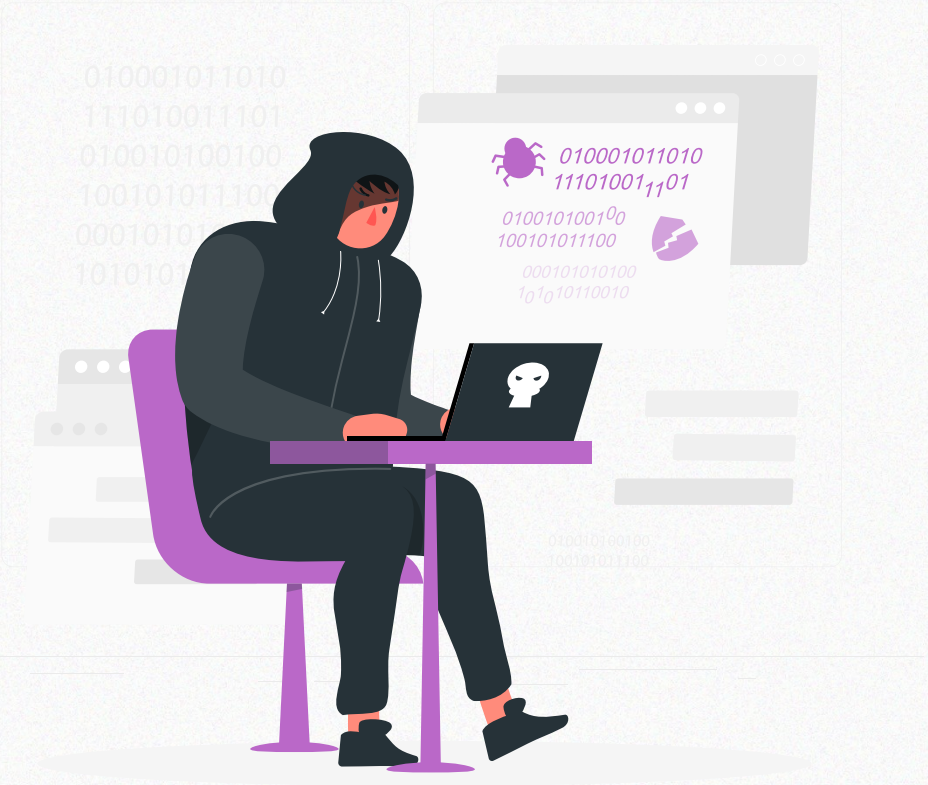
- Firmar digitalmente los componentes y dependencias distribuidos.
- Diseñar mecanismos para verificar la integridad de datos importantes.
- Implementar CI/CD seguro que valide los cambios antes del despliegue.
- Usar registros inmutables y mecanismos de auditoría.

A09:2021 - Security Logging and Monitoring Failures

Falta de monitoreo y registro de eventos críticos, lo que impide detectar y responder a ataques.

Para evitarlo un arquitecto debería:

- Diseñar un sistema de monitoreo y logging centralizado.
- Incluir logs detallados y categorizados para cada evento crítico.
- Implementar alertas automáticas para comportamientos sospechosos.
- Usar herramientas SIEM para correlación y análisis de eventos.
- Probar el sistema de monitoreo mediante ejercicios de simulación de incidentes.



A10:2021 - Server-Side Request Forgery (SSRF)

Un atacante manipula peticiones del servidor a recursos internos o externos.

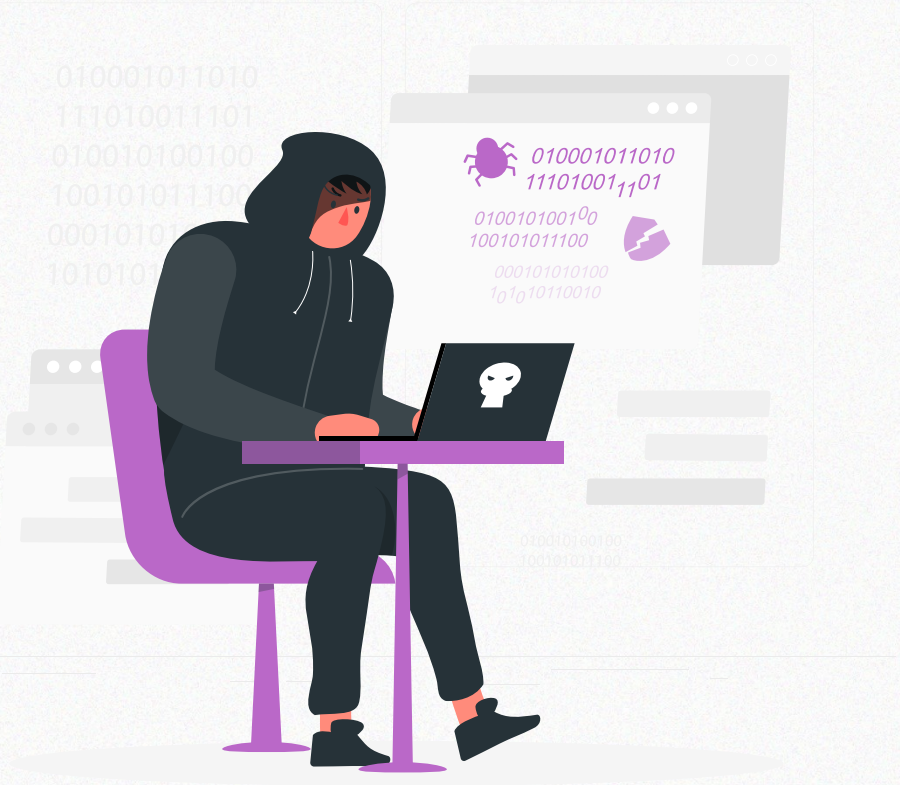
Para evitarlo un arquitecto debería:

• Validar y sanitizar todas las URLs o parámetros enviados por el usuario.

• Restringir las solicitudes del servidor a listas permitidas (allowlist).

• Implementar firewalls para bloquear peticiones no autorizadas a recursos internos.

• Diseñar sistemas que no expongan credenciales sensibles en servicios accesibles.



Patrones de Seguridad

¿Qué son los Patrones de Seguridad?

Un patrón de seguridad es una solución comprobada para un problema recurrente en el diseño de sistemas seguros.

Estos patrones sirven como guías prácticas para proteger el acceso, los datos, y la comunicación de los sistemas.

Si tuvieras que proteger la información de una empresa, ¿cómo decidirías quién puede ver, modificar o borrar cada dato?

¿Lo harías por cargo, por contexto, por características del usuario… o cómo?

Authentication and Authorization

Autenticación (Authentication):

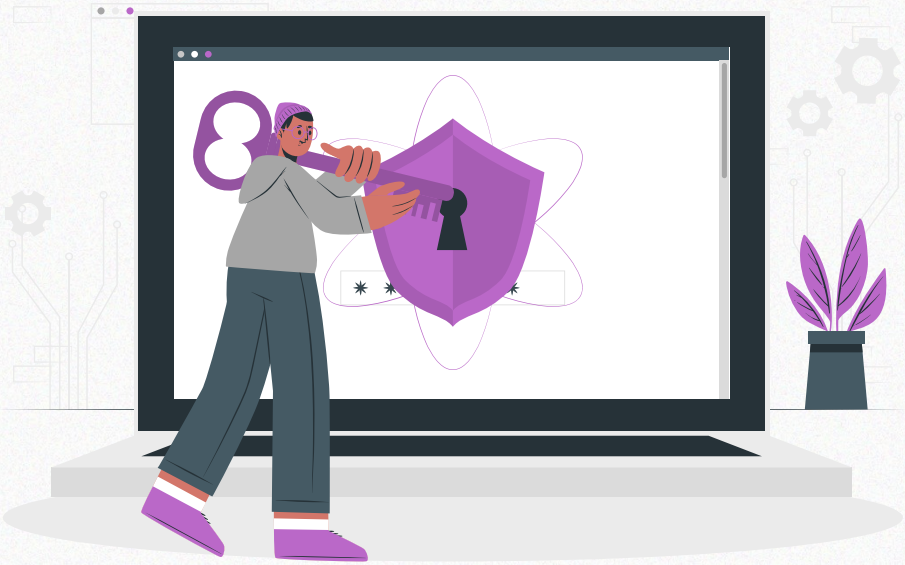
¿Quién eres?
La autenticación se enfoca en **verificar la identidad** de un usuario o sistema. Es el primer paso para establecer confianza.

Autorización (Authorization):

¿Qué puedes hacer?
La autorización **controla el acceso** a recursos o funcionalidades con base en los permisos asignados al usuario o sistema autenticado.

Métodos de Autenticación:

- **OAuth 2.0:** Protocolo estándar para delegar la autenticación entre aplicaciones y terceros.
- **OpenID Connect (OIDC):** Extensión de OAuth 2.0 para incluir autenticación de usuarios.
- **SAML (Security Assertion Markup Language):** Usado comúnmente en entornos empresariales.
- **Biometría:** Reconocimiento facial, huellas dactilares o voz, habilitado mediante SDKs como AWS Cognito o Azure AD B2C.



Métodos de Autorización:

- **Role-Based Access Control (RBAC):** Control basado en roles predefinidos, útil para sistemas jerárquicos.
- **Attribute-Based Access Control (ABAC):** Control dinámico basado en atributos del usuario y del contexto, ideal para entornos complejos.
- **Policy-as-Code:** Herramientas como **OPA (Open Policy Agent)** permiten expresar reglas de autorización de forma declarativa y reutilizable.

Role-Based Access Control (RBAC)

RBAC es un modelo donde los permisos se asignan a roles, y los usuarios heredan esos permisos al ser asignados a un rol

- Usado ampliamente en **sistemas jerárquicos**: hospitales, bancos, gobiernos.
- Permite delegar permisos y simplificar la administración.

En un hospital: un 'Médico' puede acceder a historiales clínicos, pero un 'Recepcionista' no.

Si una persona cambia de recepcionista a médico, solo cambiamos su rol.

¿Cómo resuelvo si un médico trabaja temporalmente en dos áreas con permisos diferentes?

Attribute-Based Access Control (ABAC)

ABAC permite o deniega acceso según atributos de usuarios (edad, cargo), recursos (sensibilidad de datos), acciones (leer, escribir), y contexto (hora, IP)

Dinámico y flexible, ideal para **entornos complejos o de nube**

Un analista financiero solo puede exportar datos si está en la oficina y es horario laboral; si está conectado desde casa, solo puede leerlos.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "s3:*",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/Department": "Finance"
        }
      }
    }
  ]
}
```

¿Cómo audito o explico una decisión de acceso si hay 20 atributos evaluados?

Policy-as-Code

Policy-as-Code consiste en expresar las reglas de autorización de forma declarativa, revisable y ejecutable en tiempo de decisión

Herramientas: **OPA (Open Policy Agent)**, **Rego language**, **Kubernetes admission controllers**.

```
package authz

allow {
  input.user.role == "doctor"
  input.resource.type == "medical_record"
  input.action == "read"
}
```

Esta política permite que un doctor lea historiales clínicos.

Separación de Concerns (SoC)

Como arquitectos, debemos garantizar que la lógica de autenticación y autorización esté separada de la lógica de negocio. Esto facilita la escalabilidad, el mantenimiento y la seguridad.

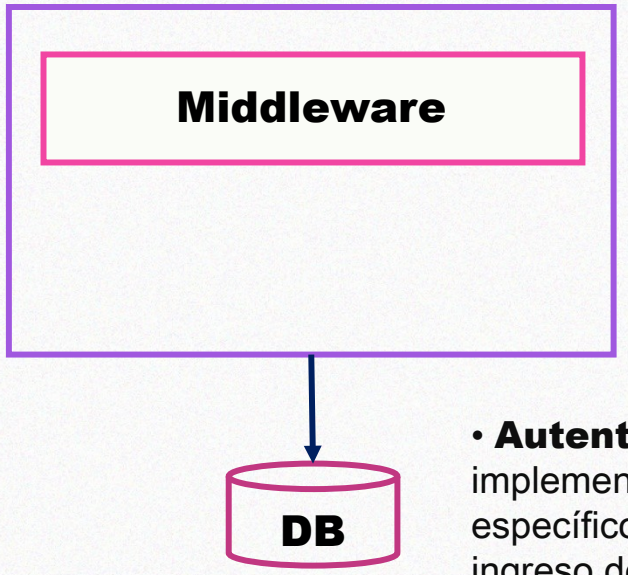
1. **Mantenibilidad:** Si surge un cambio en los requisitos de autenticación o autorización, solo necesitas modificar los módulos dedicados, no el sistema completo.

2. **Reutilización:** Los módulos o servicios de autenticación y autorización pueden ser reutilizados por diferentes aplicaciones dentro de la organización.

3. **Escalabilidad:** La SoC permite diseñar componentes escalables por separado.

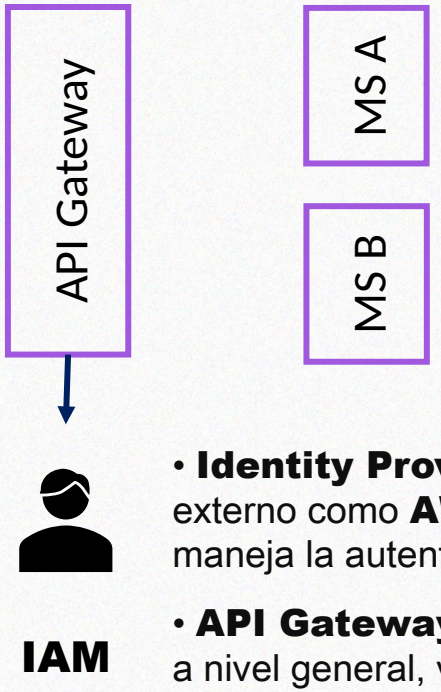
4. **Seguridad:** Centralizar la autenticación reduce la posibilidad de errores comunes relacionados con la seguridad, como almacenar credenciales en múltiples lugares.

En un monolito



- **Autenticación:** Se puede implementar en un módulo específico que gestione el ingreso de usuarios, validación de contraseñas, y generación de tokens.
- **Autorización:** Se encapsula en otro módulo que verifique permisos y roles antes de acceder a una función o recurso.

Microservicios



- **Identity Provider (IdP):** Un servicio externo como **AWS Cognito** o **Auth0** maneja la autenticación.
- **API Gateway:** Realiza la autorización a nivel general, verificando el token JWT y los claims del usuario.
- **Servicios individuales:** Cada microservicio tiene su propio nivel de autorización para validar permisos específicos.

Patrones de Seguridad en Comunicación

- **Secure Channel (Canal Seguro):** Garantiza que los datos transferidos entre dos puntos estén cifrados y autenticados.

- **Message Integrity:** Asegura que los mensajes no sean alterados durante la transmisión.

- **End-to-End Encryption (E2EE):** Los datos están cifrados desde el remitente hasta el destinatario, y solo ellos pueden descifrarlos.

¿Cuántas veces al día tienen que autenticarse en diferentes sistemas?

Patrones de Gestión de Identidad

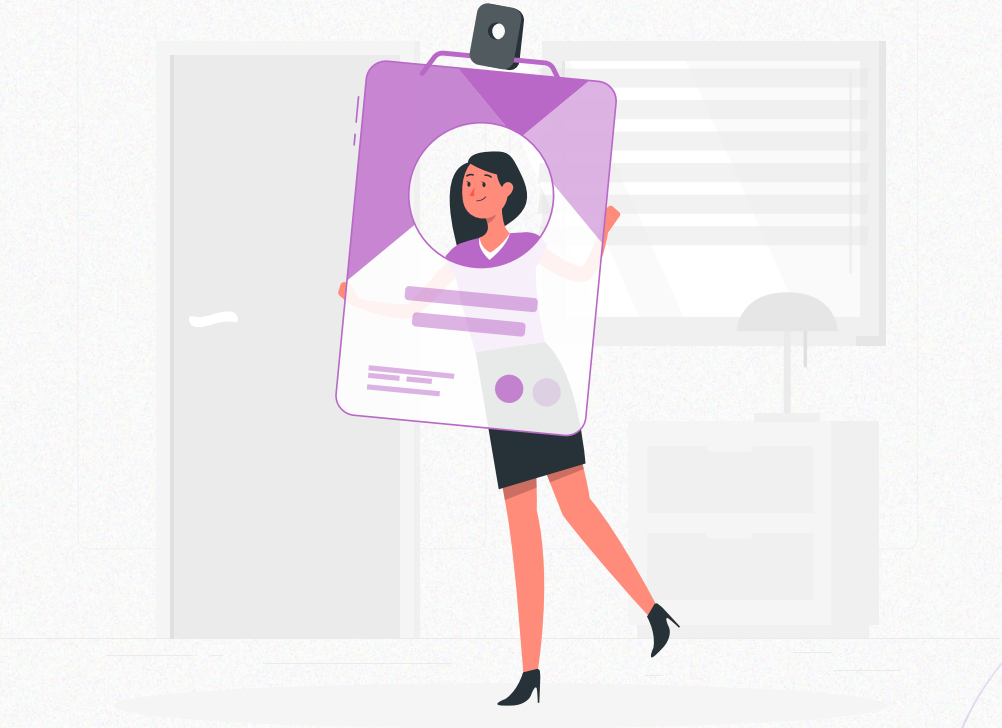
- **Federated Identity:** Permite a los usuarios autenticarse en múltiples sistemas con una sola identidad.
- **Single Sign-On (SSO):** Los usuarios se autentican una vez y acceden a múltiples sistemas.
- **Multi-Factor Authentication (MFA):** Requiere múltiples factores (algo que sabes, algo que tienes, algo que eres) para autenticar.

Federated Identity

Federated Identity permite a los usuarios usar una única identidad gestionada por un proveedor externo para autenticarse en múltiples sistemas

Flujo OIDC:

1. App redirige al usuario al IdP (Google, Okta, Keycloak).
2. Usuario se autentica en el IdP.
3. IdP retorna un **ID Token (JWT)** firmado.
4. App valida el token y otorga acceso.



Single Sign-On (SSO)

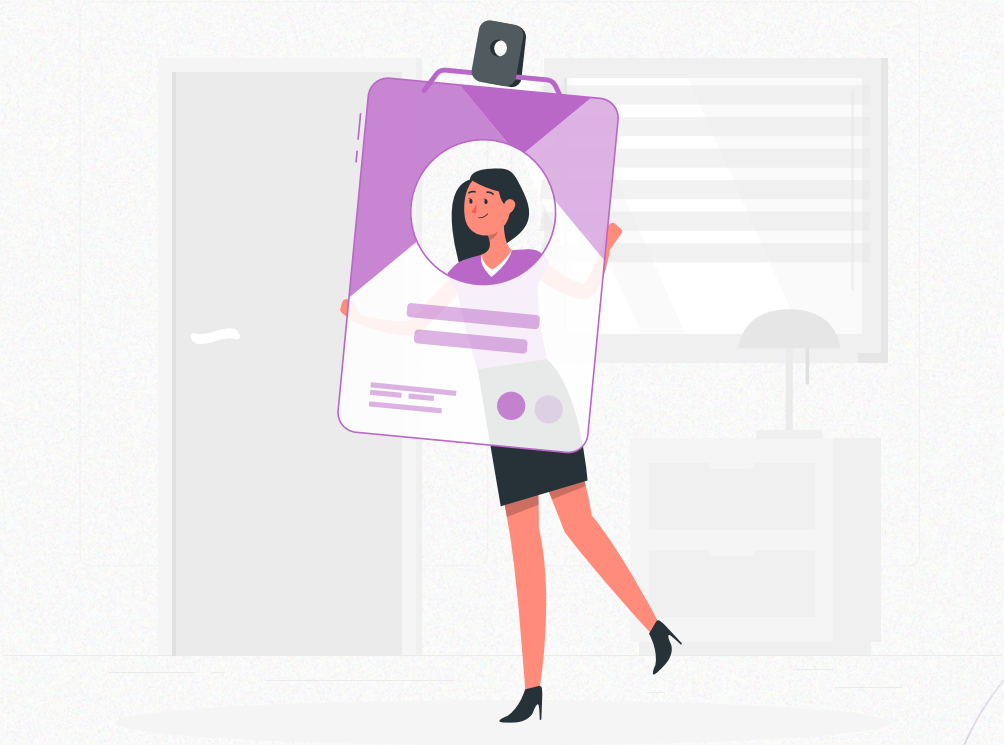
SSO permite que un usuario se autentique una sola vez y acceda a múltiples aplicaciones sin volver a ingresar sus credenciales.

Protocolos:

- **Kerberos (legacy)** – utilizado en Active Directory.
- **SAML** – login único en apps web corporativas.
- **OIDC** – moderno, basado en JWT.

Flujo (OAuth2 + OIDC):

- User login → recibe cookie de sesión o token JWT.
- Cada app verifica el token con el IdP (Keycloak, Auth0, Okta).



Multi-Factor Authentication (MFA)

MFA requiere más de un factor de autenticación: algo que sabes (contraseña), algo que tienes (token), algo que eres (biometría).

Factores comunes:

- Algo que sabes → Contraseña.
- Algo que tienes → OTP, YubiKey, push notification.
- Algo que eres → Huella digital, Face ID.

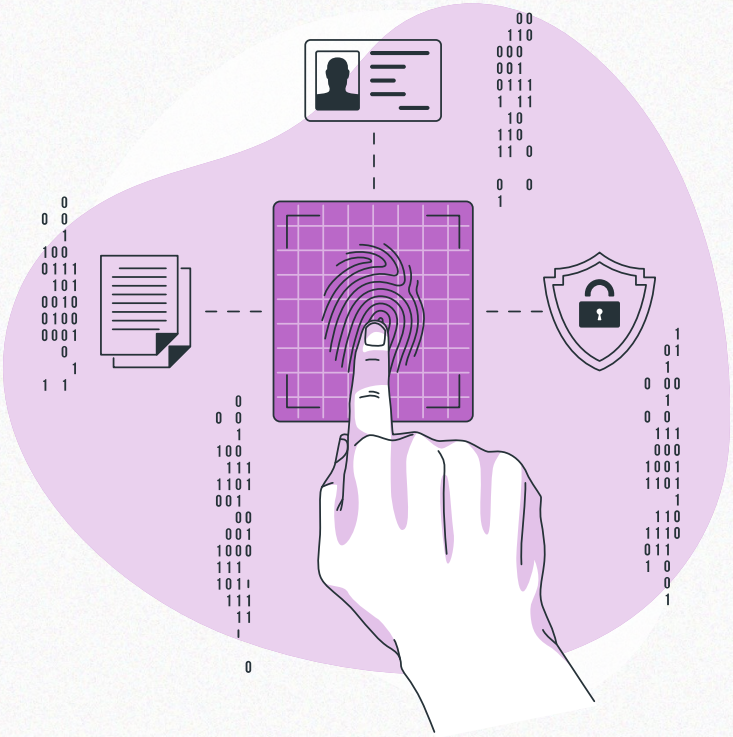
Patrones de Protección de Datos

- **Data Encryption at Rest:** Cifra los datos almacenados para protegerlos contra accesos no autorizados.
- **Data Masking:** Oculta datos sensibles mostrando versiones parcial o completamente ofuscadas.
- **Secure Key Management:** Gestiona de forma segura las claves de cifrado.

Patrones de Seguridad para Sistemas Distribuidos

- **Zero Trust Architecture:** Supone que ningún sistema o usuario es confiable por defecto, incluso dentro de la red.

- **Service Mesh:** Proporciona una capa de comunicación segura entre microservicios, manejando autenticación, autorización y cifrado.



Monitoreo y Respuesta a Incidentes

Detección de amenazas:

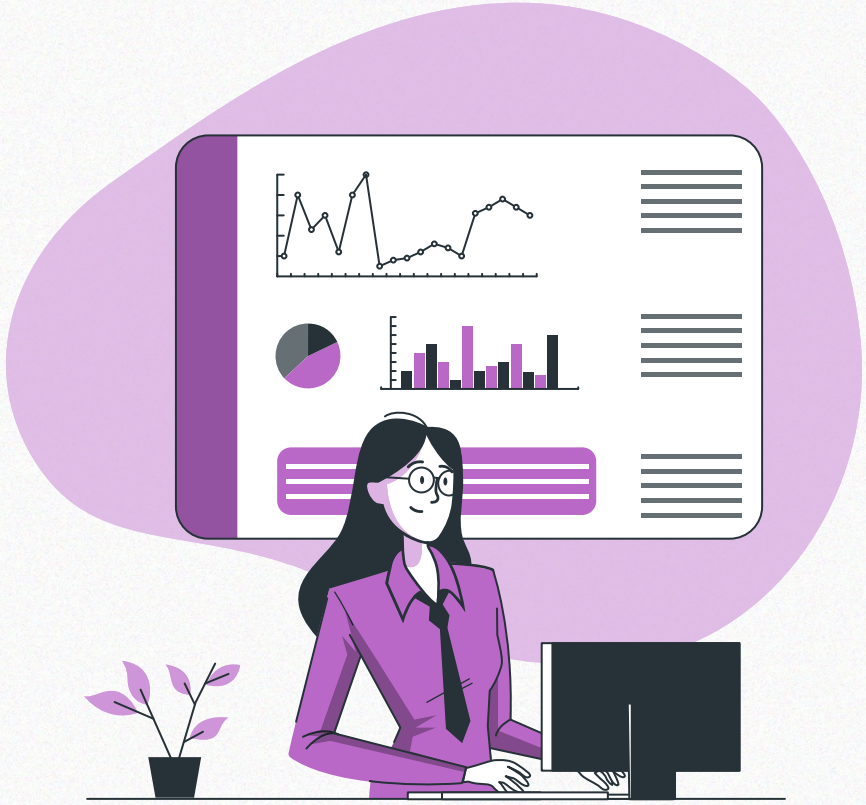
• Monitoreo de tráfico de red con herramientas como **Wireshark**, **Suricata**, o sistemas IDS/IPS.

• Análisis de logs mediante plataformas como **ELK Stack**, **Splunk**, o **Datadog**.

Automatización de respuestas:

• Implementación de sistemas SOAR (Security Orchestration, Automation, and Response) para responder automáticamente a incidentes.

• Integración con sistemas SIEM (Security Information and Event Management) para correlacionar eventos.



Monitoreo y Respuesta a Incidentes

Detección de amenazas:

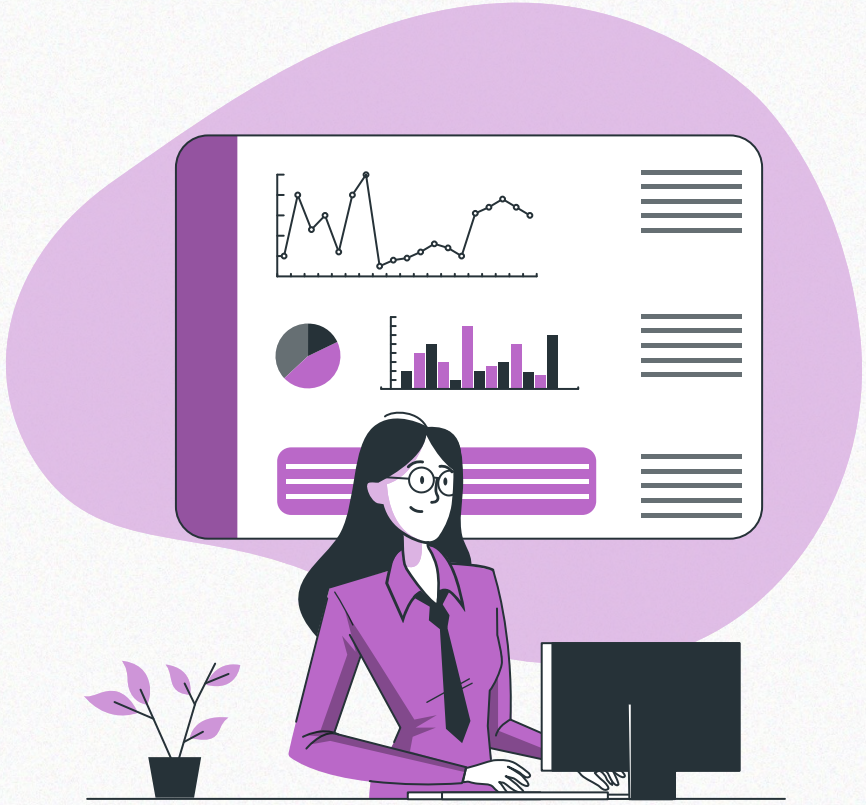
• Monitoreo de tráfico de red con herramientas como **Wireshark**, **Suricata**, o sistemas IDS/IPS.

• Análisis de logs mediante plataformas como **ELK Stack**, **Splunk**, o **Datadog**.

Automatización de respuestas:

• Implementación de sistemas SOAR (Security Orchestration, Automation, and Response) para responder automáticamente a incidentes.

• Integración con sistemas SIEM (Security Information and Event Management) para correlacionar eventos.



Conceptos Clave

Teorema CAP

El teorema CAP, propuesto por Eric Brewer en el año 2000 y formalizado en 2002, establece que un sistema distribuido no puede garantizar simultáneamente los tres aspectos siguientes en presencia de una partición de red:

C

A

P

Consistency (C): Consistencia: Todos los nodos de un sistema ven los mismos datos al mismo tiempo. Esto implica que después de realizar una operación de escritura, todas las lecturas reflejarán la última actualización.

Availability (A): Disponibilidad: Cada solicitud recibe una respuesta, ya sea con el resultado correcto o un error, pero no hay tiempo de espera indefinido. En otras palabras, el sistema está siempre disponible para aceptar solicitudes.

Partition Tolerance (P): Tolerancia a Particiones: El sistema continúa funcionando incluso si la red sufre particiones (fallos o retrasos en la comunicación entre nodos).

Ejemplos prácticos de CAP en acción

Ejemplo 1: Bases de datos relacionales distribuidas (CP)

- **Prioridad: Consistencia y Tolerancia a Particiones**
- **Caso de uso:** Sistemas financieros, donde la exactitud es crítica.
- **Trade-off:** La disponibilidad puede sacrificarse temporalmente para garantizar que los datos sean consistentes entre nodos después de una partición.

Ejemplo 2: Bases de datos NoSQL como Cassandra (AP)

- **Prioridad: Disponibilidad y Tolerancia a Particiones**
- **Caso de uso:** Redes sociales o sistemas de streaming donde la latencia es más importante que la consistencia inmediata.
- **Trade-off:** Los datos pueden estar desactualizados en algunos nodos temporalmente.

Teorema CAP

Puedes vivir en un lado del triángulo, pero no estar en las tres esquinas a la vez durante un fallo de red

Tipo	Garantiza	Ejemplo
CP	Consistencia + Partición	MongoDB (con write concern “majority”)
AP	Disponibilidad + Partición	Couchbase, Cassandra
CA	Consistencia + Disponibilidad (solo sin particiones)	RDBMS tradicionales (PostgreSQL, MySQL)

¿Qué significa construir sistemas seguros y confiables?

¿Qué tan seguros son los sistemas que usamos diariamente?

Confidencialidad

Garantizar que los datos solo son accesibles para las personas autorizadas.

Integridad

Asegurar que los datos no se alteren de manera no autorizada.

Disponibilidad

Asegurar que los servicios y datos estén accesibles cuando se necesitan.

La seguridad es un componente clave de la confiabilidad.

Un sistema confiable debe resistir fallos, ataques y errores humanos.

Importancia de la Seguridad

Costos económicos:

Según estudios recientes, el costo promedio de una brecha de seguridad puede superar los $4 millones (IBM 2023).

Pérdida de Datos Sensibles

- Exposición de datos personales de clientes o empleados (como nombres, direcciones, números de tarjetas de crédito, etc.).
- Pérdida de propiedad intelectual, secretos comerciales o información confidencial.

Consecuencias Legales y Regulatorias

- Sanciones financieras debido al incumplimiento de regulaciones como GDPR, HIPAA o CCPA.
- Posibles demandas legales por parte de clientes o socios afectados.

Prácticas Recomendadas para la Seguridad y Confiabilidad

La automatización ayuda a garantizar que las medidas de seguridad se apliquen consistentemente, sin depender de procesos manuales que son propensos a errores.

Automatización de la Seguridad

- Análisis de código estático
- Pruebas de seguridad automatizadas
- Gestión de parches automatizada
- Supervisión continua

Prácticas Recomendadas para la Seguridad y Confiabilidad

La automatización ayuda a garantizar que las medidas de seguridad se apliquen consistentemente, sin depender de procesos manuales que son propensos a errores.

Auditorías y Sistemas Auditables

- Implementar un sistema de registro de auditorías (logging)
- Auditorías de terceros
- Revisión de permisos y accesos

Prácticas Recomendadas para la Seguridad y Confiabilidad

La automatización ayuda a garantizar que las medidas de seguridad se apliquen consistentemente, sin depender de procesos manuales que son propensos a errores.

Pruebas de Resiliencia

- Simulación de fallos (Chaos Engineering)
- Pruebas de recuperación ante desastres
- Pruebas de carga y estrés

Prácticas Recomendadas para la Seguridad y Confiabilidad

La automatización ayuda a garantizar que las medidas de seguridad se apliquen consistentemente, sin depender de procesos manuales que son propensos a errores.

Gestión de Identidad y Acceso

- Autenticación multifactor (MFA)
- Principio de privilegio mínimo
- Rotación de claves y contraseñas
- Revisión de accesos inactivos

Gestión de Incidentes y Resiliencia

El ciclo de vida de un incidente de seguridad se divide en cuatro fases principales:

Detección Identificar el incidente lo más rápido posible para mitigar su impacto.

1. **Monitoreo continuo:** Implementar herramientas de monitoreo y detección como SIEMs (Splunk, ELK, Wazuh).
2. **Alertas automatizadas:** Configurar notificaciones para eventos inusuales
3. **Análisis proactivo:** Realizar análisis regulares de registros y tráfico de red.

Gestión de Incidentes y Resiliencia

El ciclo de vida de un incidente de seguridad se divide en cuatro fases principales:

Respuesta

Contener y mitigar el incidente.

1. **Equipo de respuesta a incidentes (IRT):** Definir un equipo multidisciplinario responsable de manejar incidentes.
2. **Planes de acción claros:** Crear un manual con procedimientos para incidentes comunes (DDoS, fuga de datos, ransomware).
3. **Aislamiento del sistema afectado:** Cortar la conexión de la máquina comprometida para evitar la propagación.

Gestión de Incidentes y Resiliencia

El ciclo de vida de un incidente de seguridad se divide en cuatro fases principales:

Comunicación

Notificar a las partes interesadas y minimizar el impacto reputacional.

1. **Comunicación interna:** Informar al equipo técnico y a los líderes sobre el incidente.
2. **Comunicación externa:** Notificar a los clientes si es necesario (por ejemplo, en caso de fuga de datos).

Gestión de Incidentes y Resiliencia

El ciclo de vida de un incidente de seguridad se divide en cuatro fases principales:

Recuperación Restaurar los servicios y garantizar que no vuelva a ocurrir.

1. **Revisar la causa raíz:** Analizar el incidente para identificar las vulnerabilidades que lo permitieron.
2. **Actualizar medidas de seguridad:** Aplicar parches, reforzar configuraciones y rotar credenciales.
3. **Aprendizaje continuo:** Realizar una reunión post mortem para identificar qué se hizo bien y qué se puede mejorar.

Diseñando para la Resiliencia

La capacidad de un sistema para continuar operando o recuperarse rápidamente tras un fallo.

Principios de Resiliencia

1. **Redundancia:** Diseñar sistemas con componentes redundantes para garantizar disponibilidad.

    • Ejemplo: Bases de datos replicadas en varias zonas de disponibilidad.

2. **Failover automático:** Configurar sistemas para redirigir el tráfico automáticamente a recursos saludables.

    • Ejemplo: Amazon AWS usa balanceadores de carga que detectan instancias fallidas y redirigen el tráfico.

3. **Aislamiento de fallos:** Diseñar componentes de manera que un fallo en uno no afecte a otros.

    • Ejemplo: Microservicios con límites claros en su

Diseñando para la Resiliencia

La capacidad de un sistema para continuar operando o recuperarse rápidamente tras un fallo.

Pruebas de Resiliencia:

1. **Simulación de fallos (Chaos Engineering):** Introducir fallos intencionales para verificar la capacidad del sistema para resistir.
    - Herramientas: Chaos Monkey, Gremlin.
2. **Simulacros de recuperación ante desastres:** Probar periódicamente los procedimientos de recuperación para asegurarse de que sean efectivos.
    - Ejemplo: Restaurar copias de seguridad en un entorno de prueba.

Ejemplo Real:

Gestión de un Ataque DDoS

Un sitio de comercio electrónico experimenta un ataque DDoS durante el Black Friday, lo que satura su servidor principal y afecta la experiencia del cliente.

Acciones tomadas:

1. **Detección:**

- El SIEM detecta tráfico inusual desde miles de direcciones IP.
- Se activa una alerta para el equipo de respuesta a incidentes.

2. **Respuesta:**

- El tráfico se redirige automáticamente a un sistema de mitigación DDoS (Cloudflare).
- Los ingenieros analizan los logs para identificar patrones del ataque.

3. **Recuperación:**

- Se habilitan más servidores en la infraestructura (escalado horizontal).
- Se implementan reglas en el firewall para bloquear IPs maliciosas.

4. **Lecciones aprendidas:**

- Ajustar las configuraciones del balanceador de carga para responder más rápido en futuros ataques.
- Ampliar la capacidad de la infraestructura durante eventos de alta demanda.

Integrando Seguridad al Ciclo de Desarrollo

Caso Real: Microsoft

Era el año 2001, y Microsoft estaba en la cima del mundo tecnológico. Windows XP acababa de lanzarse, y los usuarios estaban encantados con su interfaz moderna y nuevas funcionalidades. Pero detrás del brillo, los ingenieros de Microsoft enfrentaban un problema crítico: **seguridad**.

Windows XP se había convertido en un blanco fácil para los atacantes. Las críticas crecían, y cada día aparecían nuevas vulnerabilidades explotables, desde gusanos como **Blaster** hasta virus que exponían información sensible. La reputación de Microsoft como líder tecnológico estaba en juego.

Bill Gates envió un correo interno a todo el personal técnico declarando que la **seguridad debía ser la prioridad número uno**. Ese fue el punto de partida del Security Development Lifecycle (SDL).

Introducción al SDL y Fases del Ciclo de Vida

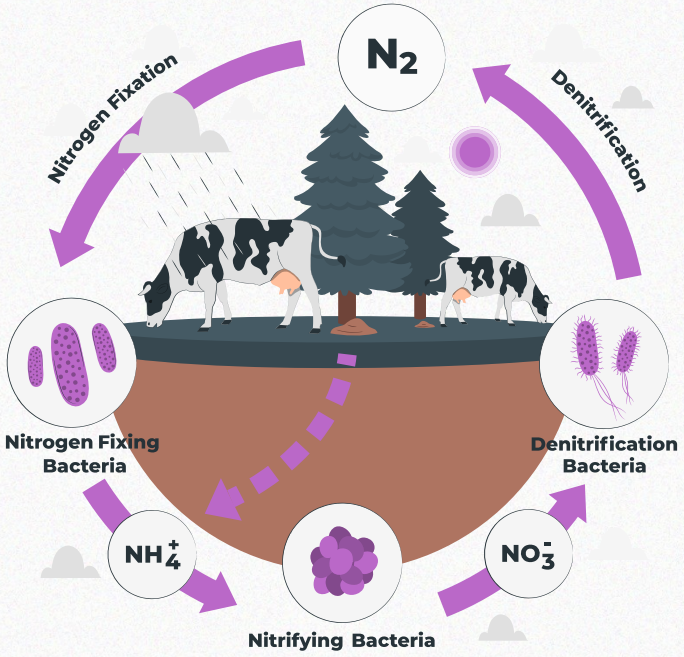
¿Qué es el SDL?

- • Un marco de trabajo para integrar la seguridad en cada etapa del desarrollo de software.
- • Creado y popularizado por Microsoft, con el objetivo de reducir vulnerabilidades y riesgos en sus productos.

Importancia del SDL

• Abordar problemas de seguridad desde el diseño, en lugar de reaccionar a ellos después de la implementación.

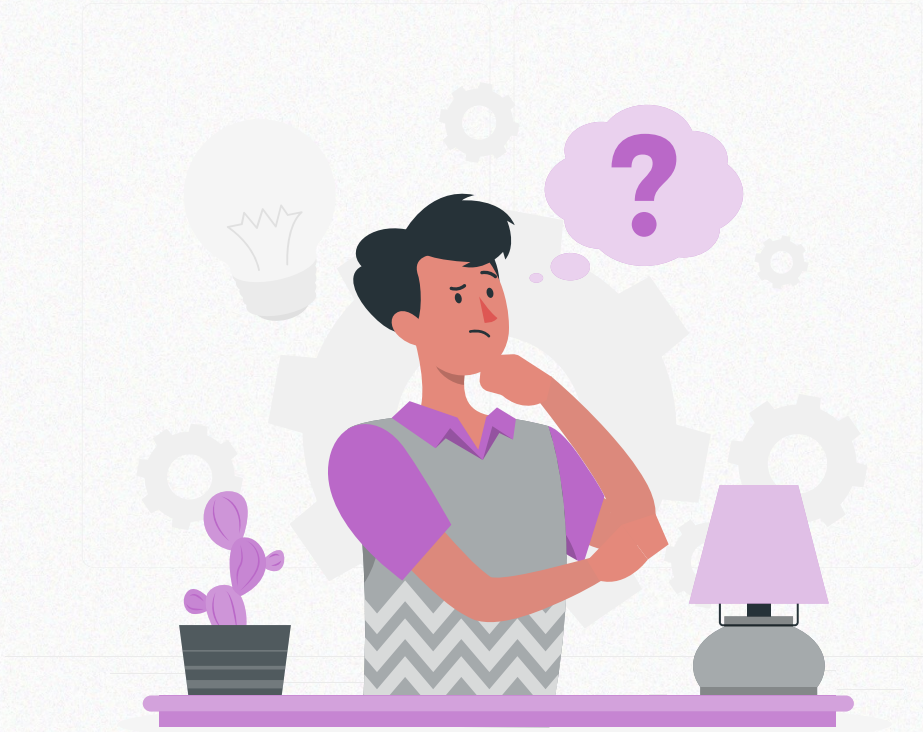
• Reducción de costos asociados con la corrección de vulnerabilidades en fases posteriores.



Principios clave del SDL:

La seguridad no es un añadido,
sino una parte integral del desarrollo.

La seguridad debe ser medible y
evaluada a lo largo del proyecto.



Fases del SDL

Fase 1 Requisitos

- Identificar requisitos de seguridad.
- Definir objetivos de seguridad claros
- Herramientas: cuestionarios de riesgo y hojas de trabajo para documentar requisitos.

Fase 2 Diseño

- **Análisis de amenazas:** Identificar y priorizar riesgos utilizando herramientas como STRIDE
- **Modelado de seguridad:** Crear diagramas que resalten flujos de datos y posibles puntos vulnerables.

Fase 3 Implementación

Buenas prácticas:

- Validar entradas de usuario.
- Evitar inyecciones de código (SQL, comandos, etc.).

Fase 4 Verificación

Pruebas de seguridad:

- Pruebas de penetración y fuzz testing.
- Verificar que las vulnerabilidades detectadas anteriormente hayan sido resueltas.

Fase 5 Lanzamiento

- Asegurarse de que las configuraciones predeterminadas sean seguras.
- Implementar registros y monitoreo en producción.

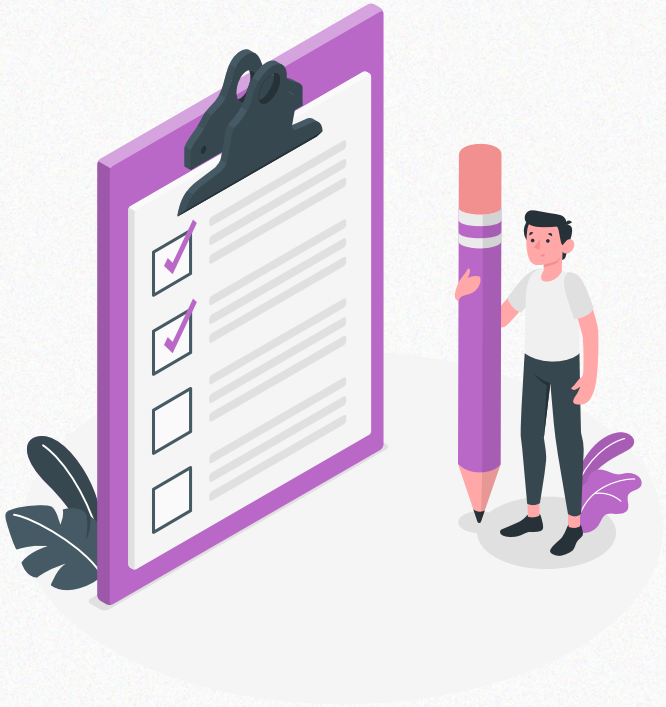
Fase 6 Respuesta

- Preparar planes de respuesta a incidentes.
- Monitorear constantemente el sistema para detectar vulnerabilidades futuras.

Implementación de SDL en Proyectos Reales

Herramientas recomendadas:

- **Requisitos:** OWASP Risk Assessment Template.
- **Diseño:** Threat Dragon, Microsoft Threat Modeling Tool.
- **Implementación:** SonarQube, Snyk.
- **Verificación:** OWASP ZAP, Nessus, Burp Suite.
- **Lanzamiento:** Ansible, Terraform para infraestructura como código segura.
- **Respuesta:** SIEMs como Splunk o ELK Stack.



Caso Microsoft: Cambiar la Cultura de Seguridad

Microsoft sabía que no podía seguir simplemente reaccionando a las amenazas. Necesitaban una estrategia estructurada para evitar problemas de seguridad desde el principio. Aquí es donde nació el SDL.

El objetivo: Introducir prácticas de seguridad en cada etapa del desarrollo del software para reducir las vulnerabilidades críticas antes de que llegaran al cliente.

Proyecto piloto: Windows XP Service Pack 2 (SP2).
En lugar de agregar nuevas características, el equipo decidió enfocarse exclusivamente en mejorar la seguridad del sistema operativo.

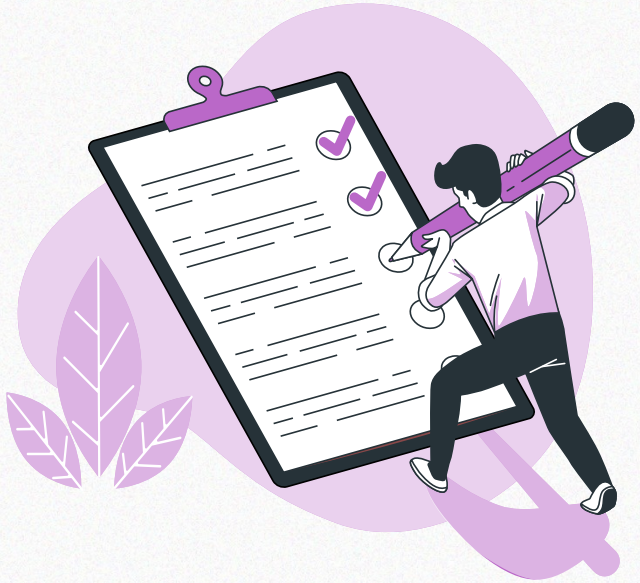
Cómo el SDL Transformó Windows XP SP 2

Requisitos: ¿Qué necesitamos proteger?

Los ingenieros comenzaron por identificar los activos más críticos de Windows XP:

- **Información del usuario:** Datos personales y credenciales de inicio de sesión.
- **Servicios críticos:** Acceso remoto, redes compartidas, puertos abiertos.
- **Aplicaciones instaladas:** Bibliotecas y servicios que podrían ser explotados.

Nueva práctica: Crear un “cuestionario de seguridad” para identificar riesgos en cada módulo del sistema operativo.



Cómo el SDL Transformó Windows XP SP 2

Diseño: Anticiparse al enemigo

El equipo implementó **modelado de amenazas** por primera vez a gran escala.

Ejemplo:

- Los ingenieros usaron **STRIDE** para analizar características clave como el firewall y la gestión de red.
- Detectaron que los puertos abiertos por defecto en Windows XP podían ser utilizados por gusanos como Blaster.

Medida tomada: Configurar el firewall de Windows XP para bloquear todos los puertos no esenciales de forma predeterminada.

Cómo el SDL Transformó Windows XP SP 2

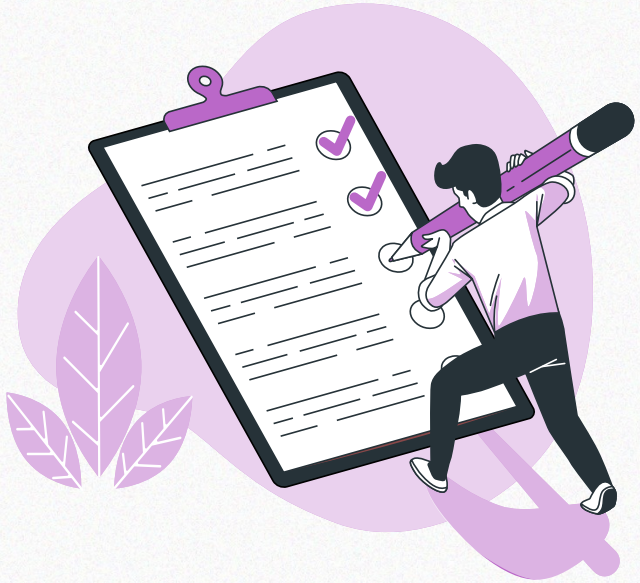
Implementación: Código Seguro

El equipo adoptó estrictas **prácticas de codificación segura**, introduciendo herramientas de análisis de código para buscar:

- Buffer overflows.
- Falta de validación de entradas.
- Exposiciones no intencionadas de datos.

Ejemplo: Una de las vulnerabilidades más comunes en las versiones anteriores era el desbordamiento de búfer.

Medida tomada: Utilizar herramientas internas como PREfix y PREfast para analizar el código y prevenir estos errores antes de compilarlo.

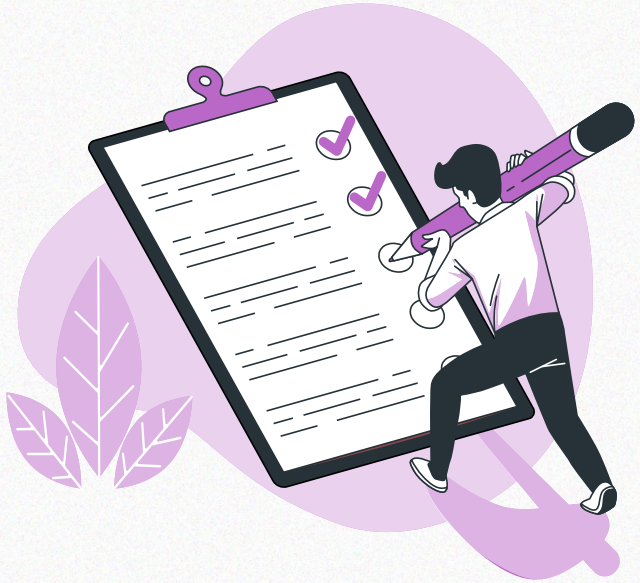


Cómo el SDL Transformó Windows XP SP 2

Verificación: Someter el sistema a prueba

Una vez implementadas las mejoras, Windows XP SP2 fue sometido a:

- **Pruebas de penetración:** Hackers éticos intentaron vulnerar las nuevas defensas del sistema.
- **Fuzz testing:** Enviar datos aleatorios a componentes críticos para detectar comportamientos inesperados.



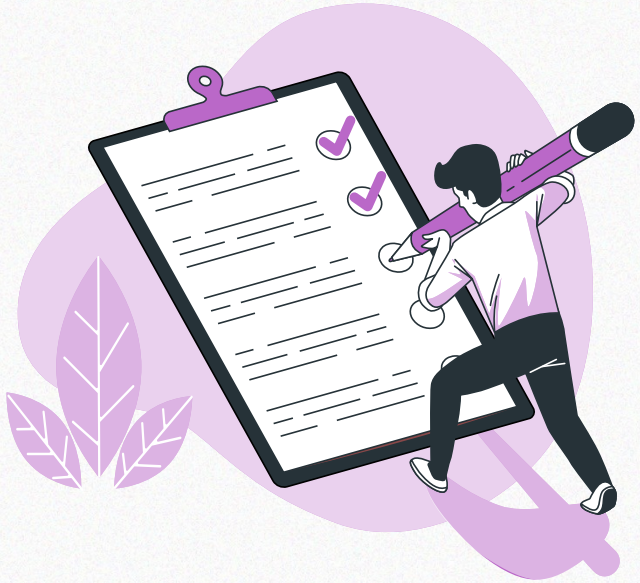
Resultado: Se descubrieron más de **1,000 vulnerabilidades** durante las pruebas, todas solucionadas antes del lanzamiento.

Cómo el SDL Transformó Windows XP SP 2

Lanzamiento: Entrega Segura

En lugar de simplemente publicar el Service Pack, el equipo incluyó configuraciones predeterminadas seguras para todos los usuarios:

- El firewall de Windows estaba **activado por defecto**, lo que redujo significativamente la exposición inicial de las máquinas recién instaladas.
- Se mejoró la protección contra scripts maliciosos en Internet Explorer.



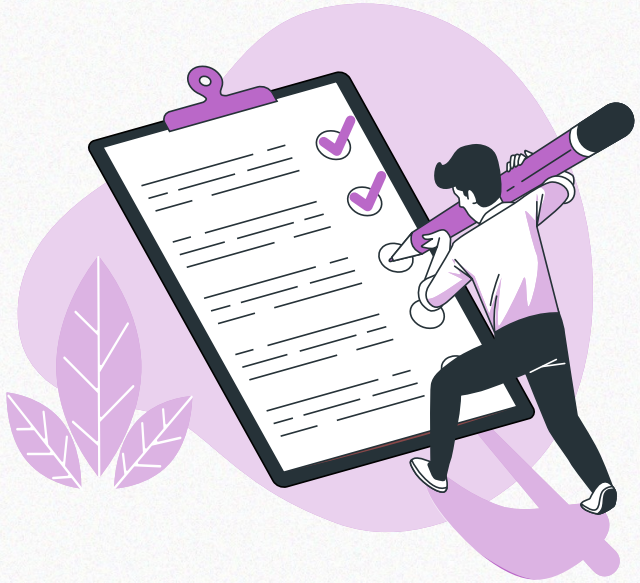
Medida extra: Documentación clara para usuarios y administradores sobre cómo aprovechar las nuevas medidas de seguridad.

Cómo el SDL Transformó Windows XP SP 2

Respuesta: Monitoreo continuo

Después del lanzamiento, Microsoft estableció un equipo de respuesta a incidentes dedicado exclusivamente a Windows XP SP2:

- Monitoreaban continuamente reportes de vulnerabilidades en todo el mundo.
- Implementaron actualizaciones automáticas para distribuir rápidamente parches de seguridad.



Fundamento de las Pruebas de Seguridad

¿Qué son las pruebas de seguridad?

- Proceso de identificar vulnerabilidades y brechas en sistemas y aplicaciones.
- Enfocadas en prevenir accesos no autorizados, garantizar la confidencialidad, integridad y disponibilidad de la información.

Tipos de pruebas de seguridad

1. **Pruebas de caja blanca**:
    - Con acceso completo al código fuente.
    - Evaluación exhaustiva, pero requiere más tiempo y experiencia técnica.
2. **Pruebas de caja negra**:
    - Sin acceso al código, simulando un ataque externo.
    - Evalúa la postura de seguridad desde la perspectiva de un atacante.
3. **Pruebas de caja gris**:
    - Acceso parcial a la información del sistema.
    - Balance entre tiempo y profundidad de análisis.

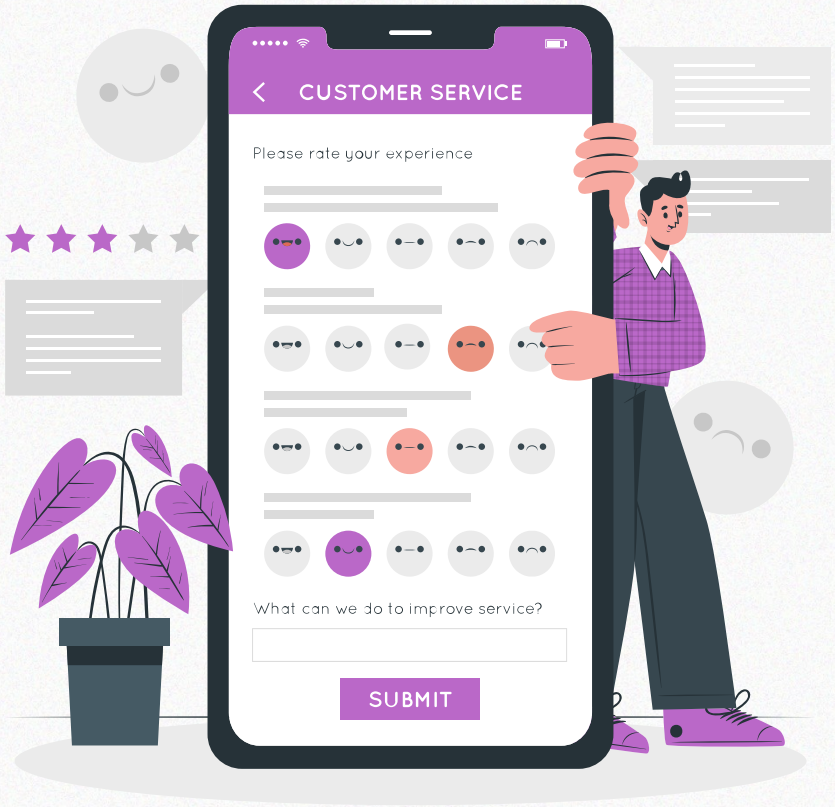
Técnicas para Pruebas de Seguridad

Evaluación de vulnerabilidades

Identificación y priorización de vulnerabilidades conocidas.

Técnicas clave:

- Escaneo automatizado de vulnerabilidades.
- Revisión manual de configuraciones críticas.



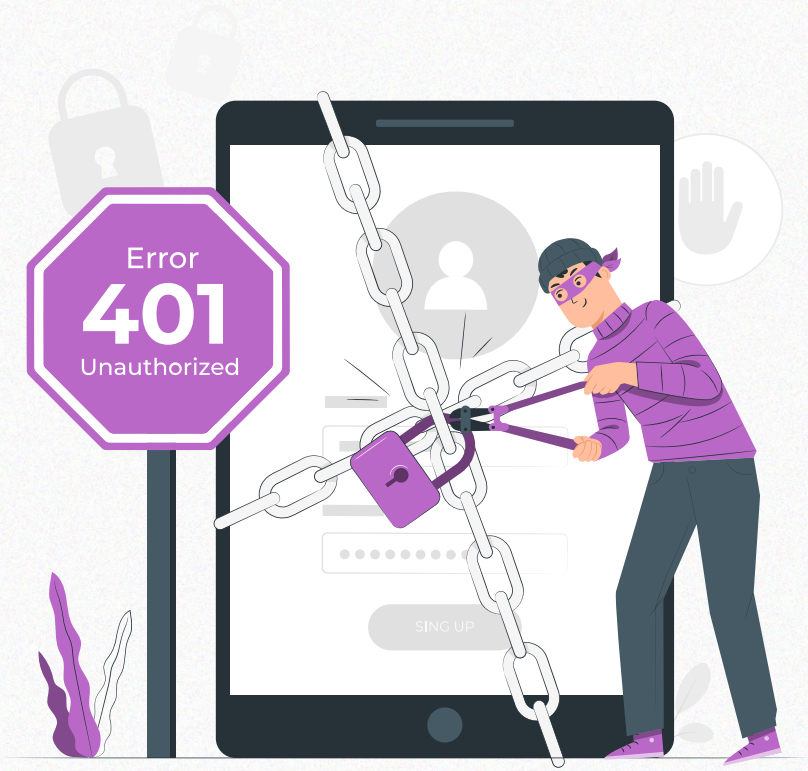
Técnicas para Pruebas de Seguridad

Penetration Testing (Pentesting)

Simula un ataque real para identificar brechas de seguridad.

Etapas:

1. Reconocimiento.
2. Escaneo.
3. Explotación.
4. Post-explotación.
5. Informe.



Pruebas de ingeniería social

Simulación de ataques a través de manipulación psicológica.

Ejemplos:

- Phishing.
- Smishing (phishing vía SMS).
- Baiting (cebos físicos como USB infectados).

Herramientas Esenciales para Pruebas de Seguridad

Escáneres de Vulnerabilidades

Los escáneres de vulnerabilidades son herramientas diseñadas para identificar posibles debilidades en sistemas, redes o aplicaciones, utilizando bases de datos actualizadas de vulnerabilidades conocidas.

Nmap (Network Mapper)

Descubrimiento de redes, escaneo de puertos y detección de servicios.

```
nmap -sS -sV -T4 -A -p- target_ip
```

Características:

- Identificación de hosts activos.
- Detección de sistemas operativos y versiones de software.
- Scripts NSE (Nmap Scripting Engine) para pruebas avanzadas.

Herramientas Esenciales para Pruebas de Seguridad

Escáneres de Vulnerabilidades

Los escáneres de vulnerabilidades son herramientas diseñadas para identificar posibles debilidades en sistemas, redes o aplicaciones, utilizando bases de datos actualizadas de vulnerabilidades conocidas.

Nessus

Escaneo de vulnerabilidades en redes y sistemas.

```
nmap -sS -sV -T4 -A -p- target_ip
```

Características:

- Análisis detallado de configuraciones y políticas.
- Identificación de vulnerabilidades con priorización basada en riesgo.
- Integración con sistemas de gestión de riesgos.

Herramientas Esenciales para Pruebas de Seguridad

Escáneres de Vulnerabilidades

Los escáneres de vulnerabilidades son herramientas diseñadas para identificar posibles debilidades en sistemas, redes o aplicaciones, utilizando bases de datos actualizadas de vulnerabilidades conocidas.

OpenVAS

Escaneo de vulnerabilidades en entornos corporativos.

```
nmap -sS -sV -T4 -A -p- target_ip
```

Características:

- Solución de código abierto.
- Base de datos constantemente actualizada.
- Enfoque en redes y configuraciones críticas.

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Pruebas de Penetración (Pentesting)

Estas herramientas simulan ataques reales para evaluar las defensas de un sistema o aplicación.

Metasploit

Desarrollo y ejecución de exploits.

```
use exploit/windows/smb/ms17_010_eternalblu
set RHOST target_ip
exploit
```

Características:

- Gran base de datos de exploits preconfigurados.
- Módulos para escaneo, explotación y post-explotación.
- Framework extensible para personalizar ataques.

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Pruebas de Penetración (Pentesting)

Estas herramientas simulan ataques reales para evaluar las defensas de un sistema o aplicación.

Burp Suite

Pruebas de seguridad en aplicaciones web.

Características:

- Interceptor de tráfico HTTP/HTTPS.
- Escáner automatizado de vulnerabilidades.
- Herramientas manuales como Intruder y Repeater.

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Pruebas de Penetración (Pentesting)

Estas herramientas simulan ataques reales para evaluar las defensas de un sistema o aplicación.

OWASP ZAP (Zed Attack Proxy)

Escaneo automatizado de aplicaciones web.

Características:

- Ideal para principiantes.
- Detección de inyecciones SQL, XSS y otras vulnerabilidades comunes.
- Integración con pipelines CI/CD.

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Pruebas de Penetración (Pentesting)

Estas herramientas simulan ataques reales para evaluar las defensas de un sistema o aplicación.

SQLmap

Automatización de pruebas de inyección SQL.

```
sqlmap -u "http://target.com?id=1" --dbs
```

- **Características**:
- Identifica y explota inyecciones SQL en bases de datos.
- Soporte para múltiples bases de datos (MySQL, PostgreSQL, Oracle).

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Auditoría de Redes

Las herramientas de auditoría de redes permiten analizar el tráfico, evaluar la configuración de seguridad y detectar actividades sospechosas.

Wireshark

Captura y análisis de tráfico de red.

Características:

- Detección de tráfico malicioso o no cifrado.
- Análisis en tiempo real.
- Filtros avanzados para paquetes específicos.

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Auditoría de Redes

Las herramientas de auditoría de redes permiten analizar el tráfico, evaluar la configuración de seguridad y detectar actividades sospechosas.

Aircrack-ng

Auditoría de redes Wi-Fi

```
airodump-ng wlan0
```

Características:

- Captura de paquetes para analizar redes cifradas.
- Ataques para descifrar claves WEP y WPA.

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Auditoría de Redes

Las herramientas de auditoría de redes permiten analizar el tráfico, evaluar la configuración de seguridad y detectar actividades sospechosas.

Ettercap

Ataques de intermediario (Man-in-the-Middle, MITM).

Características:

- Intercepción de tráfico en redes locales.
- Modificación de paquetes en tiempo real.
- Soporte para múltiples protocolos.

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Análisis de Código

El análisis de código es crucial para identificar vulnerabilidades en el nivel de desarrollo, ya sea estático (sin ejecución) o dinámico (durante la ejecución).

SonarQube

Identificación de errores y vulnerabilidades en el código.

Características:

- Soporte para múltiples lenguajes de programación.
- Integración con herramientas CI/CD.
- Detección de problemas de seguridad como inyección SQL o XSS.

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Análisis de Código

El análisis de código es crucial para identificar vulnerabilidades en el nivel de desarrollo, ya sea estático (sin ejecución) o dinámico (durante la ejecución).

Snyk

Escaneo de dependencias y bibliotecas.

Características:

- Detección de vulnerabilidades en librerías de terceros.
- Alertas de actualizaciones de seguridad.
- Integración con repositorios de código.

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Gestión de Reportes y Automatización

Estas herramientas ayudan a centralizar resultados, generar informes y automatizar flujos de trabajo en pruebas de seguridad.

Dradis

Gestión de informes de pruebas de seguridad.

Características:

- Centralización de hallazgos y reportes.
- Compatible con herramientas como Metasploit y Nessus.

Herramientas Esenciales para Pruebas de Seguridad

Herramientas para Gestión de Reportes y Automatización

Estas herramientas ayudan a centralizar resultados, generar informes y automatizar flujos de trabajo en pruebas de seguridad.

Splunk

Análisis de logs y detección de amenazas.

Características:

- Correlación de eventos de seguridad.
- Alertas en tiempo real basadas en patrones sospechosos.

Tendencias y Futuro de las Pruebas de Seguridad

A medida que las amenazas cibernéticas evolucionan, también lo hacen las estrategias y tecnologías para mitigar riesgos. En este contexto, las pruebas de seguridad están adoptando nuevas tendencias y herramientas para mantenerse a la vanguardia.

Análisis predictivo de amenazas:

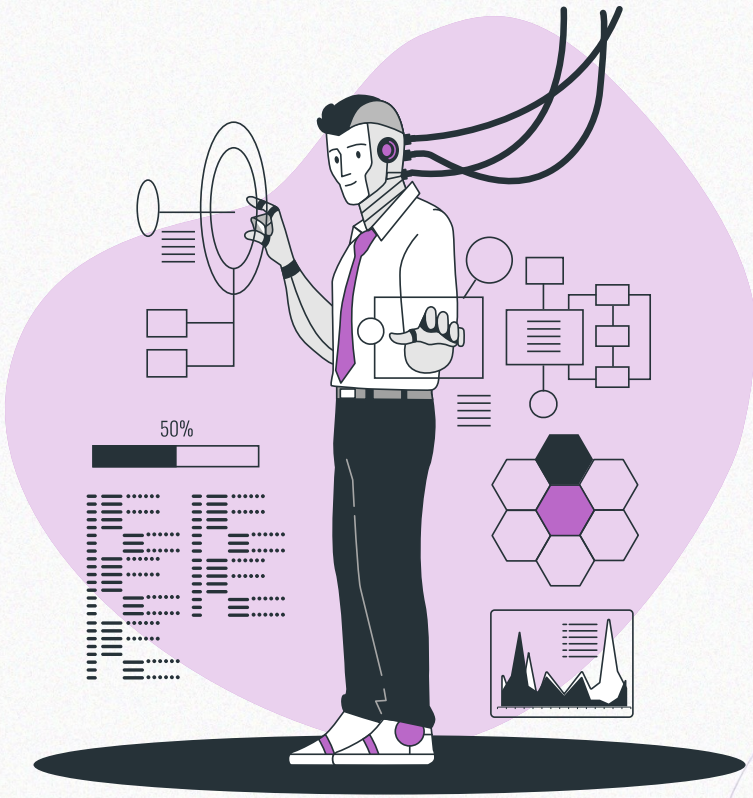
• Anticipar ataques basados en patrones históricos.

• Identificar anomalías en el comportamiento de usuarios y sistemas.

Automatización de pruebas:

• Generación dinámica de casos de prueba para encontrar vulnerabilidades desconocidas (zero-day).

• Detección de errores en configuraciones de seguridad utilizando aprendizaje automático.

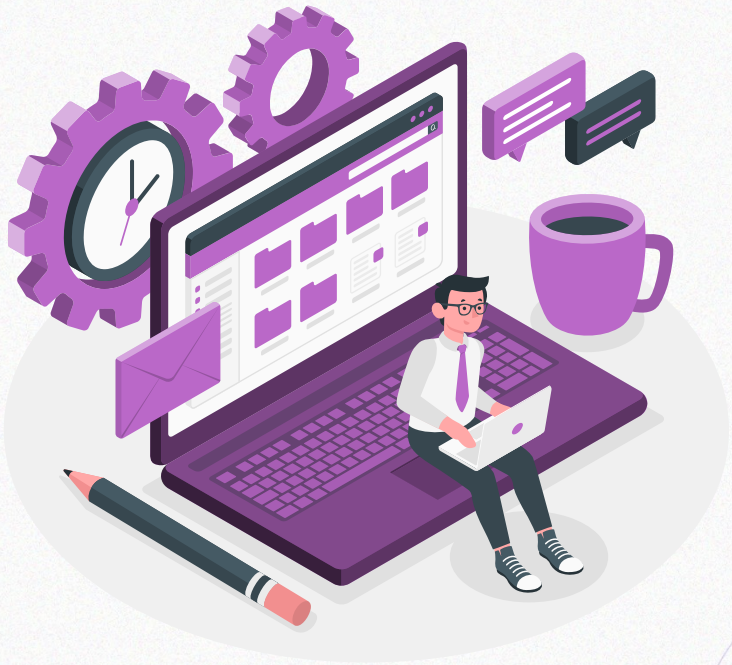


Gestionando la deudaTécnica

Qué es la Deuda Técnica y Cómo Surge

La deuda técnica es el costo acumulado de tomar decisiones de diseño y desarrollo subóptimas para cumplir con plazos o prioridades comerciales inmediatas.

- **Presión por cumplir plazos**: Atajos en el diseño, pruebas o documentación.
- **Falta de alineación técnica**: Decisiones mal informadas entre equipos de desarrollo y stakeholders.
- **Adopción de nuevas tecnologías**: Cambios en herramientas, frameworks o lenguajes que hacen obsoletos enfoques anteriores.
- **Evolución de requisitos**: Cambios en las necesidades del negocio que exigen adaptaciones constantes.

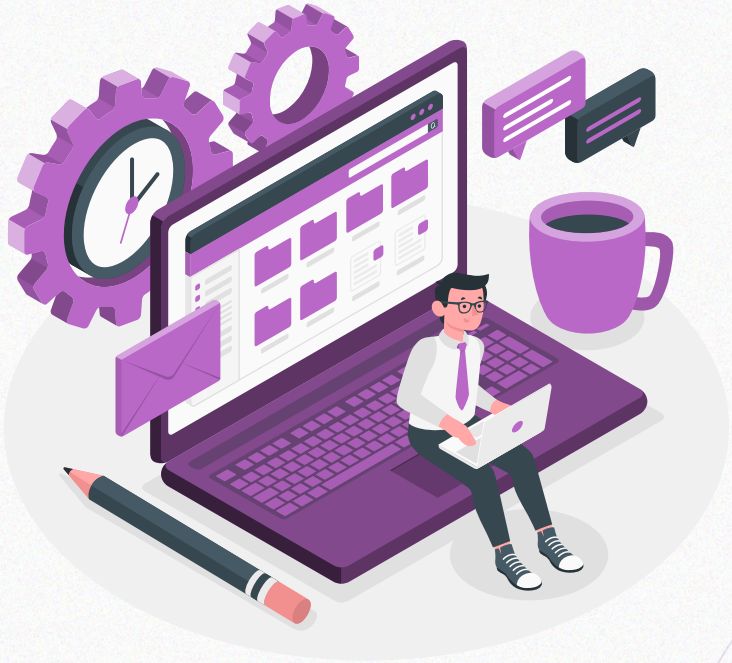


Principios Clave para Gestionar la Deuda Técnica

1. **Reconocer y priorizar la deuda técnica como un problema continuo, no ocasional.**

2. **Integrar su gestión en los ciclos de desarrollo como parte de la estrategia técnica.**

3. **Comunicar su impacto a los stakeholders no técnicos, mostrando cómo afecta el negocio.**



Estrategias para Reducir la Deuda Técnica

Gestionar la deuda técnica requiere un enfoque balanceado entre atender los compromisos del negocio y garantizar la calidad técnica a largo plazo.

Incorporar la deuda técnica en la planificación del sprint

La deuda técnica debe considerarse una parte integral del trabajo del equipo, no un esfuerzo adicional que se aborda solo cuando “hay tiempo”.

Técnica del “20% del tiempo”:

Asignar un porcentaje fijo del tiempo de cada sprint (usualmente el 10%-20%) exclusivamente para abordar deuda técnica.

> **Ejemplo:** dedicación semanal para refactorizar código, optimizar consultas SQL o mejorar la cobertura de pruebas.

• **Ventaja**: No interfiere directamente con la entrega de nuevas funcionalidades y asegura progreso continuo.

Estrategias para Reducir la Deuda Técnica

Gestionar la deuda técnica requiere un enfoque balanceado entre atender los compromisos del negocio y garantizar la calidad técnica a largo plazo.

Incorporar la deuda técnica en la planificación del sprint

La deuda técnica debe considerarse una parte integral del trabajo del equipo, no un esfuerzo adicional que se aborda solo cuando “hay tiempo”.

Separación clara de tareas:

Crear tareas específicas para mejoras técnicas en el backlog, separadas de las de nuevas características.

- Ejemplo: “Optimizar el rendimiento de la API de usuarios” se gestiona como una tarea aparte de “Crear un módulo de notificaciones”.

Estrategias para Reducir la Deuda Técnica

Gestionar la deuda técnica requiere un enfoque balanceado entre atender los compromisos del negocio y garantizar la calidad técnica a largo plazo.

Incorporar la deuda técnica en la planificación del sprint

La deuda técnica debe considerarse una parte integral del trabajo del equipo, no un esfuerzo adicional que se aborda solo cuando “hay tiempo”.

Sprints enfocados en deuda técnica:

• Planificar ocasionalmente sprints dedicados exclusivamente a abordar deuda técnica acumulada.

Cuándo usarlo: Cuando la deuda técnica está causando problemas críticos o ralentizando el desarrollo significativamente.

Estrategias para Reducir la Deuda Técnica

Gestionar la deuda técnica requiere un enfoque balanceado entre atender los compromisos del negocio y garantizar la calidad técnica a largo plazo.

Priorización basada en impacto

No toda la deuda técnica tiene el mismo nivel de impacto, por lo que priorizar correctamente es esencial para evitar un enfoque disperso.

Matriz de impacto-esfuerzo:

Evaluar tareas en dos dimensiones:

- **Impacto**: ¿Cuánto mejora el sistema al solucionar este problema?
- **Esfuerzo**: ¿Cuánto tiempo y recursos tomará resolverlo?

Priorizar las tareas de **alto impacto y bajo esfuerzo** como las primeras a abordar.

Estrategias para Reducir la Deuda Técnica

Gestionar la deuda técnica requiere un enfoque balanceado entre atender los compromisos del negocio y garantizar la calidad técnica a largo plazo.

Priorización basada en impacto

No toda la deuda técnica tiene el mismo nivel de impacto, por lo que priorizar correctamente es esencial para evitar un enfoque disperso.

Efecto en el cliente:

Enfocarse primero en solucionar problemas que afectan directamente al usuario, como:

- Latencia en aplicaciones críticas.
- Fallos recurrentes en funcionalidades clave.

Ejemplo: Si los clientes reportan errores en un módulo, priorizar su refactorización.

Estrategias para Reducir la Deuda Técnica

Gestionar la deuda técnica requiere un enfoque balanceado entre atender los compromisos del negocio y garantizar la calidad técnica a largo plazo.

Refactorización continua

La refactorización no debe ser un proyecto monumental, sino un proceso continuo integrado en el flujo de trabajo.

Micro-refactorización:

Realizar pequeños ajustes incrementales al código cada vez que se trabaja en una funcionalidad.

- Ejemplo: Optimizar una función o reducir la complejidad de un método durante la resolución de un bug.

Ventaja: No requiere tiempo adicional significativo y acumula mejoras a lo largo del tiempo.

Estrategias para Reducir la Deuda Técnica

Gestionar la deuda técnica requiere un enfoque balanceado entre atender los compromisos del negocio y garantizar la calidad técnica a largo plazo.

Refactorización continua

La refactorización no debe ser un proyecto monumental, sino un proceso continuo integrado en el flujo de trabajo.

Refactorización dirigida por pruebas:

- Implementar o mejorar pruebas antes de realizar cambios significativos.
- Esto permite detectar rápidamente si el sistema se rompe tras la refactorización.

Estrategias para Reducir la Deuda Técnica

Gestionar la deuda técnica requiere un enfoque balanceado entre atender los compromisos del negocio y garantizar la calidad técnica a largo plazo.

Refactorización continua

La refactorización no debe ser un proyecto monumental, sino un proceso continuo integrado en el flujo de trabajo.

Refactorización por módulo:

- En proyectos grandes, enfocar la refactorización en módulos específicos con mayor impacto técnico o de negocio.
- Ejemplo: Actualizar solo el módulo de autenticación sin afectar otros sistemas.

Estrategias para Reducir la Deuda Técnica

Gestionar la deuda técnica requiere un enfoque balanceado entre atender los compromisos del negocio y garantizar la calidad técnica a largo plazo.

Técnicas de “Boy Scout”

Inspirada en la regla de los Boy Scouts: “Deja siempre el lugar mejor de como lo encontraste”.

Aplicación diaria:

- Cuando un desarrollador edita una parte del código, debe mejorarla dentro de sus posibilidades.
- Ejemplo: Si actualiza una función, corregir nombres de variables poco claros o eliminar código muerto en la misma clase.

Cultura del cuidado del código:

- Establecer un estándar en el equipo donde cada miembro asume la responsabilidad de dejar el código más limpio.
- Promueve un enfoque colaborativo para reducir la deuda técnica.